SANDIA REPORT

SAND2011-5011
Unlimited Release
Printed August 2011

Supersedes SAND2010-7451
Dated January 2011

IceT Users' Guide and Reference
Version 2.1

Kenneth Moreland

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2011-5011
Unlimited Release
Printed August 2011

Supersedes SAND2010-7451
dated January 2011

IceT Users' Guide and Reference

Version 2.1

Kenneth Moreland
Data Analysis and Visualization
Sandia National Laboratories
P.O. Box 5800 MS 1323
Albuquerque, NM 87185-1323
kmorel@sandia.gov

Abstract

The Image Composition Engine for Tiles (IceT) is a high-performance sort-last parallel rendering library. In addition to providing accelerated rendering for a standard display, IceT provides the unique ability to generate images for tiled displays. The overall resolution of the display may be several times larger than any viewport that may be rendered by a single machine. This document is an overview of the user interface to IceT.

Acknowledgement

I would like to thank Brian Wylie. It was his “big ideas” that got the ball rolling on the IceT algorithms and library, and it was his continuing vision that pushed us on this path to parallel rendering.

I would also like to thank the folks at Kitware, Inc. for adopting the IceT library as the parallel rendering library for ParaView. They also maintain the IceT code repository. Without them, IceT would probably be collecting dust on a crashed RAID somewhere.

Contents

List of Figures

Chapter 1

Introduction

The Image Composition Engine for Tiles (IceT) is an API designed to enable OpenGL applications to perform Sort-Last parallel rendering on very large displays. The displays are assumed to be **tiled displays**, which are displays comprising an array of display devices that act together to form a single large display. The overall resolution of the display may be several times larger than any viewport that may be rendered by a single machine. It is also assumed that several processes in the parallel application are **display processes**. That is, their entire display window makes up part of the display.

The design philosophy behind IceT is to allow very large sets of polygons to be displayed on very high resolution displays. As such, fast frame rates are sacrificed in lieu of very scalable and very high polygon/second rendering rates. That said, there are many features in IceT that allow an application to achieve interactive rates. These include image inflation, floating viewports, active pixel encoding, and data replication. Together, these features make IceT a versatile parallel rendering application that provides near optimal parallel rendering under most data size and image size combinations. As an example, the ParaView application[1] is using IceT for all of its parallel rendering needs ranging from a desktop sized image to the world's largest tiled displays and from polygon counts ranging from 1 to 1 million (and growing).

IceT is designed to take advantage of **spatial decomposition** of the geometry being rendered. That is, it works best if all the geometry on each process is located in as small a region of space as possible. When this is true, each process usually projects geometry on only a small section of the screen. This results in less work for the compositing engine. This is of particular importance for displays with a large number of pixels.

IceT can also be used to perform sort-last parallel rendering to a single display. Such **single-tile rendering** is simply a special case of the multi-tile display IceT was designed for. Many of the optimizations done by IceT apply to the single-tile mode. Using IceT for this purpose is quite worthwhile. IceT's performance should rival that of other such software image compositors.

The rest of this document describes the use of the IceT API. There are also separate manual pages for each of the functions described here. For more details on IceT's algorithms, see:

Kenneth Moreland, Brian Wylie, and Constantine Pavlakos. "Sort-last parallel ren-

[1] http://www.paraview.org

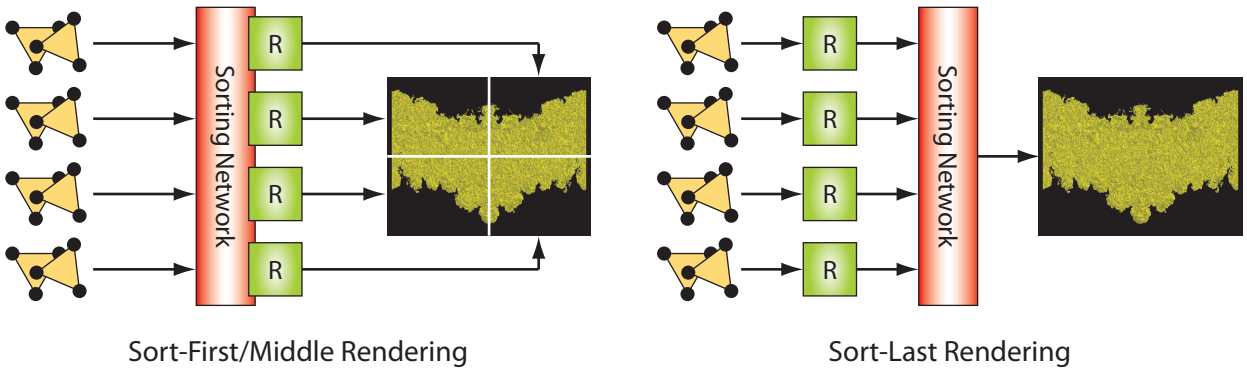

Figure 1.1. The differences between parallel rendering classes. Sort-first and sort-middle algorithms transfer geometric data. Sort-last algorithms transfer image data.

dering for viewing extremely large data sets on tile displays," In *Proceedings of IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, October 2001, pp. 85–154.

A Parallel Rendering Primer

IceT requires you to know very little about parallel rendering and their algorithms. However, it is helpful to know the basic idea behind IceT's algorithms. This section gives a brief introduction to how IceT renders in parallel.

Parallel rendering algorithms are classified as **sort-first**, **sort-middle**, or **sort-last**. The key distinguishing feature of each class is how primitives are distributed amongst processes. As demonstrated in Figure 1.1, sort-first and sort-middle algorithms allocate screen space to processes and send the appropriate geometry to each process every frame whereas sort-last algorithms render static partitions of geometry in each process and then composite the resulting images to a single image.[2] IceT is a sort-last parallel rendering library.

A convenient feature of sort-last rendering is that an application needs to change very little about how it renders geometry. The geometry is rendered the same in parallel as it is in serial; the only difference is that each process only renders a subset of the geometry. The typical operation of a parallel application using sort-last rendering is to simply render locally and then composite the images.

When rendering to a tiled display, as IceT allows you to do, there is an added level of com-

[2]In the interest of brevity and clarity, I am intentionally leaving out details that are unimportant to understanding IceT such as hybrid algorithms and differences between sort-first and sort-middle algorithms.

plexity introduced because the graphics system is often not capable of rendering an image large enough for the entire display. Thus, image compositing for a tiled display requires a loop that can iteratively render images for each tile and composite them. IceT handles this looping and interfaces with the rendering functions of your application through a callback mechanism. This will be described in the following chapters.

Chapter 2

Tutorial

In this chapter we outline the steps required to create a simple IceT application from building the IceT source, using the created libraries, and writing your own applications. IceT is solely responsible for the image composition part of parallel rendering. Thus, it relies on separate systems for rendering and communication. The two most common libraries for these features are **OpenGL** and **MPI** (the Message Passing Interface), respectively. IceT has support libraries for directly using these two systems and we will use them for this tutorial.

This tutorial assumes the reader is familiar with OpenGL or some similar rendering system. If this is your first experience with OpenGL programing, consider trying some typical serial rendering before jumping into the parallel rendering domain. A familiarity with MPI is also helpful.

Building IceT

The IceT build process is very portable. It is regularly compiled on Microsoft Windows, Macintosh OS X, and a wide variety of Unix implementations. IceT can be built with any OpenGL 1.1 compliant installation. Most modern operating systems come distributed with OpenGL. For those that are not, you can usually use the **Mesa 3D** library (www.mesa3d.org), a software implementation of OpenGL. An installation of MPI is also almost always needed, although not strictly required. **OpenMPI** (http://www.open-mpi.org/) and **MPICH** (http://www.mcs.anl.gov/mpi/mpich2/) are two free and widely portable implementation of MPI.

IceT uses **CMake** to build across so many different platforms. As such, you will have to download the CMake build tools from www.cmake.org and install. Then, create a build directory and run the CMake program (from the "Start" menu on Windows or ccmake on Unix and Mac OS X). CMake will determine the parameters of your system and do its best to find libraries on which IceT depends. The CMake program, shown in Figure 2.1 will also provide a GUI to allow you to easily change build parameters and external libraries.

CMake will generate a set of build files for the local system. The type of files depends on the type of machine you are using and the compile system you have chosen to use. On Unix machines, make files are the most common. On Windows, you usually generate MSVC project files or nmake files. On Mac OS X, either make files or Xcode project files are commonly generated based on

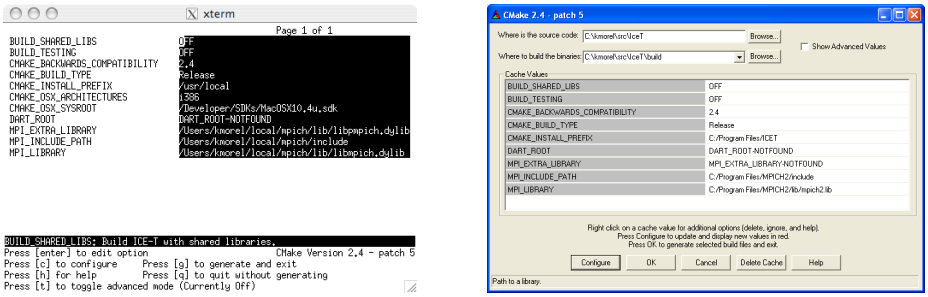

Figure 2.1. The CMake user interface. The Unix version is on the left whereas the Microsoft Windows version is on the right.

user selection. You then use the native build system to build and, optionally, install IceT.

Linking to IceT Libraries

IceT comes with three libraries: **IceTCore**, **IceTGL**, and **IceTMPI**. The actual filenames of these libraries varies depending on the filesystem and build type. For example, on most Unix systems, a static build results in filenames of libIceTCore.a and the like whereas shared libraries are libIceTCore.so. Windows has libraries with names like IceTCore.lib as well as IceTCore.dll if building shared libraries. However, the difference in these filenames usually hidden by the build system, especially if you use a portable build system like CMake.

You are, of course, free to use whatever build system you like, whether it be system specific or cross platform. Using IceT is simply a matter of finding the header and library files. However, because IceT is built with CMake, it comes with some extra facilities for helping other CMake builds find it. This section will give you the bare minimum you need to set up CMake to build an application using IceT. Readers interested in learning more about CMake should pick up a copy of *Mastering CMake* by Ken Martin and Bill Hoffman.

You define a build system with CMake by creating a **CMakeLists.txt** file. The CMakeLists.txt file is basically a simple script that gives commands to CMake to tell it how to build your project. Most CMakeLists.txt files start with the `PROJECT` command, which associates a name with your project and optionally specifies a language.

```
PROJECT(IceT_Tutorial)
```

To use IceT from within your CMake project, run the `FIND_PACKAGE` command. This command instructs CMake to find the IceTConfig.cmake file, which is written to IceT's build or install directory and contains all the necessary build settings.

```
FIND_PACKAGE(IceT REQUIRED)

INCLUDE_DIRECTORIES(${ICET_INCLUDE_DIRS})
```

Assuming that CMake finds IceT, the CMake variable **ICET_INCLUDE_DIRS** is defined and can be passed to the INCLUDE_DIRECTORIES CMake command. The variables **ICET_CORE_LIBS**, **ICET_GL_LIBS**, and **ICET_MPI_LIBS** are also defined and can be used with a TARGET_LINK_-LIBRARIES command to link in the respective IceT libraries as described later.

Any application using IceT will probably also be using OpenGL and MPI. In addition, the example in the following section also uses GLUT for window management. CMake comes with modules to find all three of these libraries, which makes it easy to include in our project.

```
FIND_PACKAGE(OpenGL REQUIRED)
FIND_PACKAGE(GLUT REQUIRED)
FIND_PACKAGE(MPI REQUIRED)

MARK_AS_ADVANCED(CLEAR
  MPI_INCLUDE_PATH
  MPI_LIBRARY
  MPI_EXTRA_LIBRARY
  )

INCLUDE_DIRECTORIES(
  ${OPENGL_INCLUDE_DIR}
  ${MPI_INCLUDE_PATH}
  ${GLUT_INCLUDE_DIR}
  )
```

The only thing left to do is to tell CMake to build a program from a set of sources and libraries specified with the ADD_EXECUTABLE and TARGET_LINK_LIBRARIES commands, respectively.

```
ADD_EXECUTABLE(Tutorial Tutorial.c)
TARGET_LINK_LIBRARIES(Tutorial
  ${OPENGL_LIBRARIES}
  ${GLUT_LIBRARIES}
  ${MPI_LIBRARY}
  ${MPI_EXTRA_LIBRARY}
  ${ICET_CORE_LIBS}
  ${ICET_GL_LIBS}
  ${ICET_MPI_LIBS}
  )
```

Creating IceT Enabled Applications

To use IceT, include its header: `IceT.h`. If you are using OpenGL for rendering, you will probably also want to use IceT's OpenGL integration functions in `IceTGL.h`. You will almost always need to also include the header containing an MPI version of an IceT communicator: `IceTMPI.h`. On the rare occasion that you need to use IceT with a communication layer other than MPI, you can define a custom communicator as described in Chapter 7.

```
#include <IceT.h>
#include <IceTGL.h>
#include <IceTMPI.h>
```

Before you call any IceT functions, you need to initialize MPI by calling `MPI_Init`. You will also need to create an **OpenGL context**. In other words, you need to make the rendering window in which the OpenGL rendering commands will go. The process for doing this is greatly dependent on the windowing system and beyond the scope of this document. It is usually easiest to use a third party API to do this. If you are not already using a GUI tool that generates OpenGL windows for you, then the **GLUT** API is a popular choice for simple applications.

You will then need to initialize IceT itself. Do this by first creating an IceT **communicator** from an MPI communicator and then using that to create an **IceT context**. When using OpenGL, you also need to initialize the OpenGL-specific code in IceT by calling **`icetGLInitialize`**.

```
comm = icetCreateMPICommunicator(MPI_COMM_WORLD);
context = icetCreateContext(comm);
icetGLInitialize();
```

In the proceeding code, `comm` is of type `IceTCommunicator` and `context` is of type `IceTContext`.

Now that we have created and activated an IceT communicator, as well as initialized the IceT **state**, we can start using IceT. It is often useful to first query IceT on the size of the parallel job it is running in and what is the local process id, or **rank**. The values are stored in variables of type `IceTInt`.

```
icetGetIntegerv(ICET_RANK, &rank);
icetGetIntegerv(ICET_NUM_PROCESSES, &num_proc);
```

Before rendering, we need to tell IceT the layout of the tiled display using the **`icetResetTiles`** and **`icetAddTile`** functions. These commands must be executed with the same arguments on all processes of the parallel job. IceT will assume that you setup the same display layout everywhere.

If you are not actually driving a tiled display and instead just generating a desktop-sized image, the following commands will correctly establish the IceT state (assuming `WINDOW_WIDTH` and `WINDOW_HEIGHT` correctly reflect the desired image dimensions).

```
icetResetTiles();
icetAddTile(0, 0, WINDOW_WIDTH, WINDOW_HEIGHT, 0);
```

The **`icetResetTiles`** function simply tells IceT that you are about to define a display layout. Each call to **`icetAddTile`** defines a tile in the display. In the case of a single image, the **single-tile rendering** mode, **`icetAddTile`** is called only once. The first two arguments to **`icetAddTile`** have no effect in this mode. The third and fourth arguments are the width and height of the image to create. Usually you set this to the width and the height of the display window, but the Image Inflation section in Chapter 4 describes other usage for these parameters. The final argument is the rank of the **display process**. After a rendering the final complete image will available only on this process. In the example above, we have directed the image to go to process zero, often referred to as the **root process**.

To define an actual tiled display, simply call the **`icetAddTile`** function multiple times. When describing tiles in a display, the first two arguments of **`icetAddTile`** describe where the lower left corner of the tile is located in respect to the overall display. All together, the first four arguments specify a viewport for the tile in an a single, cohesive high resolution display (which is what we are trying to achieve with our tiled display). The code below defines a 2×2 tiled display with the top two tiles displayed by processes 0 and 1 and the bottom two tiles displayed by processes 2 and 3.

```
icetResetTiles();
icetAddTile(0,           WINDOW_HEIGHT, WINDOW_WIDTH, WINDOW_HEIGHT, 0);
icetAddTile(WINDOW_WIDTH,WINDOW_HEIGHT, WINDOW_WIDTH, WINDOW_HEIGHT, 1);
icetAddTile(0,           0,             WINDOW_WIDTH, WINDOW_HEIGHT, 2);
icetAddTile(WINDOW_WIDTH,0,             WINDOW_WIDTH, WINDOW_HEIGHT, 3);
```

IceT contains several **strategies** for image composition. Changing the strategy modifies the algorithm IceT uses for parallel image compositing. You need to tell IceT which strategy to use with the **`icetStrategy`** function. The code below sets IceT to use the **reduce strategy**, which has proven to be an all-around good performer.

```
icetStrategy(ICET_STRATEGY_REDUCE);
```

However, when rendering only a single tile, your best bet is to use the **sequential strategy**, which bypasses some of the collective communication necessary for other strategies.

```
icetStrategy(ICET_STRATEGY_SEQUENTIAL);
```

Chapter 5 gives more detailed descriptions and advice about the strategies. Like with the display set up, all processes must set the same strategy.

IceT is almost ready to go. We just need to tell it some minimal information about how to render your geometry. First, IceT needs to know the spatial extent of the geometry to be drawn (in object space). The most natural way to do this is to use the **icetBoundingBox** function, which defines an axis-aligned box defined by the minimum and maximum coordinates in each dimension.

```
icetBoundingBoxf(x_min, x_max, y_min, y_max, z_min, z_max);
```

The parameters can, and should be, different on each process, since each process will have a different partition of data. Strictly speaking, identifying the geometry bounds is not necessary. If they are not defined, IceT will assume the geometry covers the entire screen. When rendering a single small image, the information is of little consequence. However, when rendering larger images this information can dramatically improve the performance of image composting. Specifying the bounds can be critical on large tile displays.

The second and final piece of information IceT needs is a way to draw your geometry. IceT achieves this through a **drawing callback**.

```
icetGLDrawCallback(drawScene);
```

The drawing callback is a pointer to any function that issues OpenGL commands that render geometry to the active frame buffer. The callback is free to issue most OpenGL commands so long as it restores all the OpenGL state (except, of course, frame buffer contents). Also, the callback function should modify neither the projection matrix nor the clear color. Care needs to be taken if the callback modifies the model view matrix. More details are given in the Drawing Callback section of Chapter 3.

IceT is now ready to render. Rendering is initiated with a call to **icetGLDrawFrame**. The **icetGLDrawFrame** must be called on all processes. The function will render the scene using the provided drawing callback, composite the image, and place the appropriate images in the back OpenGL buffers of the appropriate display processes.

```
icetGLDrawFrame();
```

Parallel rendering is now enabled in your application. Simply call **icetGLDrawFrame** every time you wish to draw a new image. The geometry rendered by your may change from frame to frame so long as you ensure that you also update IceT with the bounds of your geometry if it changes.

The following code is a full example of a simple IceT application. Do not be alarmed by the length. The majority of the code is spent in setting up the supporting libraries (OpenGL, GLUT, and MPI) and in comments.

```
/* -*- c -*- ************************************************************
** Copyright (C) 2007 Sandia Corporation
** Under the terms of Contract DE-AC04-94AL85000, there is a non-exclusive
** license for use of this work by or on behalf of the U.S. Government.
** Redistribution and use in source and binary forms, with or without
** modification, are permitted provided that this Notice and any statement
** of authorship are reproduced on all copies.
**
** This is a simple example of using the IceT library.  It demonstrates the
** techniques described in the Tutorial chapter of the IceT User's Guide.
************************************************************************/

#include <stdlib.h>

/* IceT does not come with the facilities to create windows/OpenGL contexts.
 * we will use glut for that. */
#ifndef __APPLE__
#include <GL/glut.h>
#include <GL/gl.h>
#else
#include <GLUT/glut.h>
#include <OpenGL/gl.h>
#endif

#include <IceT.h>
#include <IceTGL.h>
#include <IceTMPI.h>

#define NUM_TILES_X 2
#define NUM_TILES_Y 2
#define TILE_WIDTH 300
#define TILE_HEIGHT 300

static void InitIceT();
static void DoFrame();
static void Draw();

static int winId;
static IceTContext icetContext;

int main(int argc, char **argv)
{
  int rank, numProc;
  IceTCommunicator icetComm;

  /* Setup MPI. */
  MPI_Init(&argc, &argv);
```

```
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numProc);

  /* Setup a window and OpenGL context.  Normally you would just place all the
   * windows at 0, 0 (and probably full screen in tile display mode) to a local
   * display, but since this is an example we are assuming that they are all
   * going to one screen for display. */
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH | GLUT_ALPHA);
  glutInitWindowPosition((rank%NUM_TILES_X)*(TILE_WIDTH+10),
                         (rank/NUM_TILES_Y)*(TILE_HEIGHT+50));
  glutInitWindowSize(TILE_WIDTH, TILE_HEIGHT);
  winId = glutCreateWindow("IceT Example");

  /* Setup an IceT context.  Since we are only creating one, this context will
   * always be current. */
  icetComm = icetCreateMPICommunicator(MPI_COMM_WORLD);
  icetContext = icetCreateContext(icetComm);
  icetDestroyMPICommunicator(icetComm);

  /* Prepare for using the OpenGL layer. */
  icetGLInitialize();

  glutDisplayFunc(InitIceT);
  glutIdleFunc(DoFrame);

  /* Glut will only draw in the main loop.  This will simply call our idle
   * callback which will in turn call icetGLDrawFrame. */
  glutMainLoop();

  return 0;
}

static void InitIceT()
{
  IceTInt rank, num_proc;

  /* We could get these directly from MPI, but it's just as easy to get them
   * from IceT. */
  icetGetIntegerv(ICET_RANK, &rank);
  icetGetIntegerv(ICET_NUM_PROCESSES, &num_proc);

  /* We should be able to set any color we want, but we should do it BEFORE
   * icetGLDrawFrame() is called, not in the callback drawing function.
   * There may also be limitations on the background color when performing
   * color blending. */
  glClearColor(0.2f, 0.5f, 0.1f, 1.0f);
```

```
/* Give IceT a function that will issue the OpenGL drawing commands. */
icetGLDrawCallback(Draw);

/* Give IceT the bounds of the polygons that will be drawn.  Note that
 * we must take into account any transformation that happens within the
 * draw function (but IceT will take care of any transformation that
 * happens before icetGLDrawFrame). */
icetBoundingBoxf(-0.5f+rank, 0.5f+rank, -0.5, 0.5, -0.5, 0.5);

/* Set up the tiled display.  Normally, the display will be fixed for a
 * given installation, but since this is a demo, we give two specific
 * examples. */
if (num_proc < 4)
  {
  /* Here is an example of a "1 tile" case.  This is functionally
   * identical to a traditional sort last algorithm. */
  icetResetTiles();
  icetAddTile(0, 0, TILE_WIDTH, TILE_HEIGHT, 0);
  }
else
  {
  /* Here is an example of a 4x4 tile layout.  The tiles are displayed
   * with the following ranks:
   *
   *               +---+---+
   *               | 0 | 1 |
   *               +---+---+
   *               | 2 | 3 |
   *               +---+---+
   *
   * Each tile is simply defined by grabing a viewport in an infinite
   * global display screen.  The global viewport projection is
   * automatically set to the smallest region containing all tiles.
   *
   * This example also shows tiles abutted against each other.
   * Mullions and overlaps can be implemented by simply shifting tiles
   * on top of or away from each other.
   */
  icetResetTiles();
  icetAddTile(0,          TILE_HEIGHT, TILE_WIDTH, TILE_HEIGHT, 0);
  icetAddTile(TILE_WIDTH, TILE_HEIGHT, TILE_WIDTH, TILE_HEIGHT, 1);
  icetAddTile(0,          0,           TILE_WIDTH, TILE_HEIGHT, 2);
  icetAddTile(TILE_WIDTH, 0,           TILE_WIDTH, TILE_HEIGHT, 3);
  }

/* Tell IceT what strategy to use.  The REDUCE strategy is an all-around
```

```
   * good performer. */
  icetStrategy(ICET_STRATEGY_REDUCE);

  /* Set up the projection matrix as you normally would. */
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(-0.75, 0.75, -0.75, 0.75, -0.75, 0.75);

  /* Other normal OpenGL setup. */
  glEnable(GL_DEPTH_TEST);
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  if (rank%8 != 0)
    {
    GLfloat color[4];
    color[0] = (float)(rank%2);
    color[1] = (float)((rank/2)%2);
    color[2] = (float)((rank/4)%2);
    color[3] = 1.0;
    glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, color);
    }
}

static void DoFrame()
{
  /* In this idle callback, we do a simple animation loop and then exit. */
  static float angle = 0;

  IceTInt rank, num_proc;

  /* We could get these directly from MPI, but it's just as easy to get them
   * from IceT. */
  icetGetIntegerv(ICET_RANK, &rank);
  icetGetIntegerv(ICET_NUM_PROCESSES, &num_proc);

  if (angle <= 360)
    {
    /* We can set up a modelview matrix here and IceT will factor this
     * in determining the screen projection of the geometry.  Note that
     * there is further transformation in the draw function that IceT
     * cannot take into account.  That transformation is handled in the
     * application by deforming the bounds before giving them to
     * IceT. */
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glRotatef(angle, 0.0, 1.0, 0.0);
    glScalef(1.0f/num_proc, 1.0, 1.0);
```

```
    glTranslatef(-(num_proc-1)/2.0f, 0.0, 0.0);

    /* Instead of calling Draw() directly, call it indirectly through
     * icetGLDrawFrame().  IceT will automatically handle image compositing. */
    icetGLDrawFrame();

    /* For obvious reasons, IceT should be run in double-buffered frame
     * mode.  After calling icetGLDrawFrame, the application should do a
     * synchronize (a barrier is often about as good as you can do) and
     * then a swap buffers. */
    glutSwapBuffers();

    angle += 1;
    }
  else
    {
    /* We are done with the animation.  Bail out of the program here.  Clean
     * up IceT and the other libraries we used. */
    icetDestroyContext(icetContext);

    glutDestroyWindow(winId);

    MPI_Finalize();

    exit(0);
    }
}

static void Draw()
{
  IceTInt rank, num_proc;

  /* We could get these directly from MPI, but it's just as easy to get them
   * from IceT. */
  icetGetIntegerv(ICET_RANK, &rank);
  icetGetIntegerv(ICET_NUM_PROCESSES, &num_proc);

  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  /* When changing the modelview matric in the draw function, you must be
   * wary of two things.  First, make sure the modelview matrix is restored
   * to what is was when the function is called.  Remember, the draw
   * function may be called multiple times and transformations may be
   * commuted.  Also, the bounds of the drawn geometry must be correctly
   * transformed before given to IceT.  IceT has no way of knowing about
   * transformations done here.  It is an error to change the projection
   * matrix in the draw function. */
```

```
  glMatrixMode(GL_MODELVIEW);
  glPushMatrix();
  glTranslatef((float)rank, 0, 0);
  glutSolidSphere(0.5, 100, 100);
  glPopMatrix();
}
```

Chapter 3

Basic Usage

In this chapter we describe in greater detail the basic features of IceT. The tutorial given in Chapter 2 is a good place to start building your applications. You can then consult this chapter and later ones for more details on the operations as well as descriptions of further features.

Prototypes for the majority of IceT types, functions, and identifiers can be found in the `IceT.h` header file. If you are using OpenGL for rendering, which is common, you will probably want to include the header `IceTGL.h`. You will also almost always need to include the header `IceTMPI.h`. Chapter 7 provides more details on this last header file's function.

```
#include <IceT.h>
#include <IceTGL.h>
#include <IceTMPI.h>
```

The State Machine

The IceT API borrows many concepts from OpenGL. One major concept taken is that of a state machine. At all times IceT maintains a current state. The state can influence the operations that IceT makes, and IceT's operations can modify the state.

IceT can manage multiple collections of state at the same time. It does this by associating each state with a **context**. At any given time, there is at most one active context. Any IceT function called works using the current active context.

Contexts are created and destroyed with **`icetCreateContext`** and **`icetDestroyContext`**, respectively.

```
IceTContext icetCreateContext( IceTCommunicator comm );

void icetDestroyContext( IceTContext context );
```

These functions work with an object of type **`IceTContext`**. **`IceTContext`** is an opaque type; you are not meant to directly access it. Instead, you pass the object to functions to do the work for you.

The **icetCreateContext** function requires an object of type **IceTCommunicator**. This is another opaque type that is described in more detail in Chapter 7. For now, just know that you can create one from an MPI communicator using the **icetCreateMPICommunicator** function.

```
IceTCommunicator icetCreateMPICommunicator(
                                          MPI_Comm mpi_comm );

void icetDestroyMPICommunicator( IceTCommunicator comm );
```

Also be aware that if you plan to use IceT's OpenGL layer, you will need to initialize it with **icetGLInitialize**. You can query whether the OpenGL layer has be initialized with **icetGLIsInitialized**.

```
void icetGLInitialize( void );

IceTBoolean icetGLIsInitialized( void );
```

It is good practice to call **icetGLInitialize** immediately after creating a context with **icetCreateContext** to always ensure that the OpenGL layer is ready to be used (assuming you plan to use it).

The following code gives the common boilerplate for setting up your initial IceT context.

```
#include <IceT.h>
#include <IceTGL.h>
#include <IceTMPI.h>

int main(int argc, char **argv)
{
  IceTCommunicator icetComm;
  IceTContext icetContext;

  /* Setup MPI. */
  MPI_Init(&argc, &argv);

  /* Setup an IceT context.  If we are only creating one, this context will
   * always be current. */
  icetComm = icetCreateMPICommunicator(MPI_COMM_WORLD);
  icetContext = icetCreateContext(icetComm);
  icetDestroyMPICommunicator(icetComm);

  /* Initialize the OpenGL layer. */
  icetGLInitialize();

  /* Start your parallel rendering program here. */

  /* Cleanup IceT and MPI. */
```

```
  icetDestroyContext(icetContext);
  MPI_Finalize();

  return 0;
}
```

Any number of contexts may be created, each with its own associated state. At any given time, a single given context is **current**. All IceT operations are applied with the state attached to the current context. A handle to the current IceT context can be retrieved with the **icetGetContext** function, and he current context can be changed by using the **icetSetContext** function.

```
IceTContext icetGetContext( void );

void icetSetContext( IceTContext context );
```

Changing the context is a fast and easy way to swap states. This could be used, for example, to switch between rendering modes. One context could be used for a full resolution image, and another could use **image inflation** (described in Chapter 4) to make faster but coarser images during interaction.

When a context is created, its state is initialized to default values. You can effectively "duplicate" a context by copying the state of one context to another using the **icetCopyState** function.

```
void icetCopyState( IceTContext       dest,
                    const IceTContext src );
```

The state of a context comprises a group of key/value pairs. The state can be queried by using any of the **icetGet** functions.

```
void icetGetDoublev( IceTEnum     pname,
                     IceTDouble * params );

void icetGetFloatv( IceTEnum    pname,
                    IceTFloat * params );

void icetGetIntegerv( IceTEnum   pname,
                      IceTInt *  params );

void icetGetBooleanv( IceTEnum      pname,
                      IceTBoolean * params );

void icetGetPointerv( IceTEnum    pname,
                      IceTVoid ** params );
```

The valid keys that can be used in the **icetGet** functions are listed in the **icetGet** documentation starting on page 161. There is no way to directly set these state variables. Instead, they are set either by IceT configuration functions or indirectly as part of the operation of IceT. The

documentation for **icetGet** also describes which functions can be used to set each state entry (assuming the user has control of that state entry).

There is a special set of state entries that toggle IceT options. Although you can query this state with the **icetGetBooleanv** function, it is more typical to use the **icetIsEnabled** function. Also unlike the other state variables, these variables can be directly manipulated with the **icetEnable** and **icetDisable** functions.

```
IceTBoolean icetIsEnabled( IceTEnum pname );

void icetEnable ( IceTEnum pname );

void icetDisable ( IceTEnum pname );
```

The options queried with **icetIsEnabled** and manipulated with **icetEnable** and **icetDisable** are listed in the **icetEnable** documentation starting on page 158.

Diagnostics

The IceT library has a mechanism for reporting diagnostics. There are three levels of diagnostics. **Errors** are anomalous conditions that IceT considers a critical failure. An occurrence of an error generally means that the future IceT operations will have undefined behavior. When IceT is compiled in debug mode, a seg fault is intentionally raised when an error occurs to make it easier to attach a debugger to the point where the error occurred.

Warnings are detections of anomalous conditions that are not as severe as errors. When a warning occurs, the current operation may produce the incorrect results, but future operations should continue to work.

IceT also can also provide a large volume of **debug** messages. These messages simply indicate the status of IceT operations as they progress. They are generally of no use to anyone who is not trying to develop or debug IceT operations.

IceT diagnostics are controlled with the **icetDiagnostics** function.

```
void icetDiagnostics( IceTBitField mask );
```

The **icetDiagnostics** function takes a set of flags that can me or-ed together. The diagnostics for errors, warnings, and debug statements can be set by passing the **ICET_DIAG_ERRORS**, **ICET_DIAG_WARNINGS**, and **ICET_DIAG_DEBUG** flags, respectively. Turning on warnings implicitly turns on errors and turning on debug statements implicitly turns on errors and warnings (although there is no problem with redundantly specifying these flags).

IceT has the ability to report diagnostics either on all processes or only on the **root process** (the process with rank 0). This behavior is controlled by the **ICET_DIAG_ROOT_NODE** and **ICET_-**

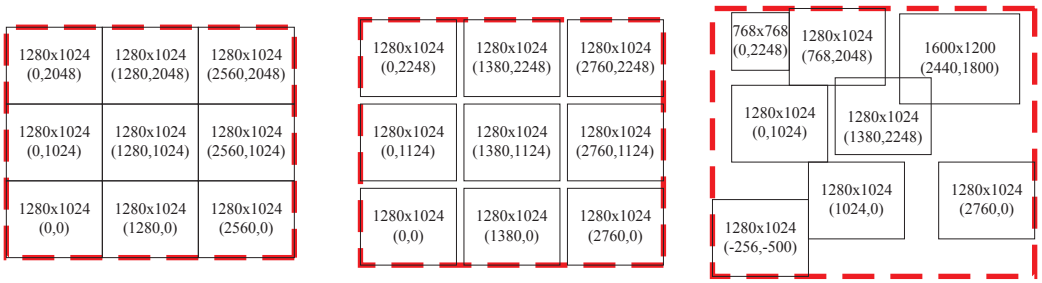

Figure 3.1. Defining a tile display with viewports in a logical global display. Three possible tile arrangements are shown. The bounds of each tile is drawn with the viewport given inside. The viewable area is shown with a dashed line.

`DIAG_ALL_NODES` flags. Many diagnostic messages occur on all nodes when they occur, so reporting only on node 0 can greatly reduce the number of messages with which to contend. However, messages can differ between processes or may not occur on all processes.

The special flags **`ICET_DIAG_FULL`** and **`ICET_DIAG_OFF`** turn all possible diagnostics on and all diagnostics completely off, respectively.

By default, IceT displays errors and warnings on all nodes. You can get the current diagnostic level by calling **`icetGet`** with **`ICET_DIAGNOSTIC_LEVEL`**.

Display Definition

IceT assumes that the tiled display it is driving has each tile connected to the graphics output of one of the processes in the parallel job in which it is running. This type of arrangement is natural for any tiled display driven by a graphics cluster, and is the delivery method of many graphics APIs.

IceT defines the configuration of a tiled display by using a **logical global display** with an infinite [1] number of pixels in both the horizontal and vertical directions. The definition of each tile comprises the identifier for the process connected to the physical projection and the **viewport** (position and size) of the tile in the global display. IceT implicitly defines the rectangle that tightly encompasses all of the tile viewports as the viewable area and snaps the viewing region (defined by the OpenGL viewing matrices) to this area.

Figure 3.1 shows some possible tile arrangements. Mullions or overlaps of the tiles in the physical display can be represented by the spacing or overlap of the viewports in the logical display.

[1] Well, OK. The logical global display only stretches as far as the 32-bit numbers that are used to define viewports. But that's still way bigger than any physical display that we can possibly conceive, so conceptually we call it infinite.

IceT does not require the tile layout to have any regularity. Chaotic layouts like that shown in the right image of Figure 3.1 are legal, although probably not very useful. It is allowed, and in fact encouraged, to have processes that are not directly connected to the tiled display. These **non-display** processes still contribute to the image compositing work and will reduce the overall time to render an image.

The display is defined using the **icetResetTiles** and **icetAddTile** functions. Any previous tile definition is first cleared out using **icetResetTiles** and new tiles are added, one at a time, using **icetAddTile**.

```
void icetResetTiles( void )

int icetAddTile( IceTInt       x,
                 IceTInt       y,
                 IceTSizeType  width,
                 IceTSizeType  height,
                 int           display_rank );
```

Each tile is specified using screen coordinates in the logical global display: the position of the lower left corner and the width and height of the tile. Each tile also has a **display process** associated with it. After an image is completely rendered and composited, the screen section belonging to this tile will be placed in the process at the given rank.

The following code demonstrates a common example for establishing the tile layout: a grid of projectors. The arrangement of projectors in this example assume that the projectors are connected to processes in the order of left to right and then top to bottom, which is common. Note, however, that IceT defines its logical global display with y values from the bottom up like OpenGL does.

```
icetResetTiles();
for (row = 0; row < num_tile_rows; row++) {
    for (column = 0; column < num_tile_columns; column++) {
        icetAddTile(column*TILE_WIDTH, (num_tile_rows-row-1)*TILE_HEIGHT,
                    TILE_WIDTH, TILE_HEIGHT,
                    row*num_tile_columns + column);
    }
}
```

Mullions are added by simply spacing the tiles apart from each other in the logical global display. Because they are defined in the logical global display, physical dimensions of the mullions must first be converted to pixels using the dot pitch of the displays. The following code adds mullions between all of the tiles.

```
icetResetTiles();
for (row = 0; row < num_tile_rows; row++) {
    for (column = 0; column < num_tile_columns; column++) {
```

```
        icetAddTile(column*(TILE_WIDTH + x_mullion),
                    (num_tile_rows-row-1)*(TILE_HEIGHT + y_mullion),
                    TILE_WIDTH, TILE_HEIGHT,
                    row*num_tile_columns + column);
    }
}
```

An equally common use for IceT is to render images in parallel to a single display. In this **single-tile rendering** mode, we simply create a single tile whose image will be placed in the GUI of some application. This is done by either using the OpenGL context of the GUI as part of the IceT rendering process or by grabbing the image of the single tile and copying into the GUI. The example code below sets up IceT to create a single image that is accessible on the root process.

```
icetResetTiles();
icetAddTile(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT, 0);
```

IceT indexes the tiles in the order that they are defined with **icetAddTile**. You can get the current definition of the tile display from a number of state variables, which can be retrieved as always with **icetGet**. **ICET_NUM_TILES** stores the number of tiles that are defined (the number of times **icetAddTile** was called). **ICET_TILE_VIEWPORTS** stores an array with all of the dimensions of each tile. For each tile, **ICET_TILE_VIEWPORTS** contains the four values $\langle x, y, width, height \rangle$, stored consecutively, corresponding to the values passed to **icetAddTile**. **ICET_DISPLAY_NODES** stores an array giving the rank of the display process displaying that tile. Each process can also query the **ICET_TILE_DISPLAYED** variable to see which tile is displayed locally. **ICET_TILE_DISPLAYED** is set to -1 on every process that does not display a tile.

You can get information about the display geometry as a whole through **ICET_GLOBAL_VIEWPORT**. This variable stores the four-tuple $\langle x, y, width, height \rangle$. x and y are placed at the left-most and lowest position of all the tiles, and *width* and *height* are just big enough for the viewport to cover all tiles.

The IceT image compositor remains decoupled from the rendering system. Calling **icetAddTile** will not create a display context or renderable window for the tile. That responsibility is left to the calling application. When using IceT in single-tile rendering mode, the rendering system should be set to produce images of that single tile's size. When driving a physical tiled display, each display process must create a window that covers the entire display. It is also a good idea to disable the mouse cursor in these windows.

Note that the size of the tiles do not have to match each other. Also, the size of the images that your application generates do not have to match the size of any of the tiles. There is, however, a constraint that the generated images on all processes must be at least as large as the largest tile in each dimension. To help you maintain that constraint, IceT stores the largest tile dimensions in the **ICET_TILE_MAX_WIDTH** and **ICET_TILE_MAX_HEIGHT** state variables.

IceT must know in advanced the size of images that your application will render. If you are using IceT's OpenGL layer, IceT will automatically know the size of the images you generate based

off of the current OpenGL viewport (retrieved with the **GL_VIEWPORT** OpenGL state variable). Otherwise, you can specify the size of images the application generates with **icetPhysicalRenderSize**. If you are not using the OpenGL layer and you have not called **icetPhysicalRenderSize**, IceT assumes that you will generate images of width **ICET_TILE_MAX_WIDTH** and height **ICET_TILE_MAX_HEIGHT**. The actual expected rendered image size is stored in the **ICET_PHYSICAL_RENDER_WIDTH** and **ICET_PHYSICAL_RENDER_HEIGHT** state variables.

Although counterintuitive, it is often more efficient to create images that are larger than any tile. This situation may be necessary when using image inflation (see Chapter 4). Even when not using image inflation, larger rendered images can save a significant amount of rendering time. IceT can use the larger images to potentially render in one shot an object that is larger than any of the tiles.

Strategies

IceT contains several algorithms for performing image compositing. The overall algorithm used to render and composite an image is called a strategy, named after the "Gang of Four" strategy pattern.[2] The strategy is set using the **icetStrategy** function.

```
void icetStrategy( IceTEnum strategy );
```

IceT defines the following strategies that can be passed to **icetStrategy**. These strategies are discussed in more detail in Chapter 5.

ICET_STRATEGY_SEQUENTIAL Basically applies a "traditional" single tile composition (such as binary swap) to each tile in the order they were defined. Because each process must take part in the composition of each tile regardless of whether they draw into it, this strategy is usually inefficient when compositing for more than one tile, but is recommended for the single tile case because it bypasses some of the communication necessary for the other multi-tile strategies.

ICET_STRATEGY_DIRECT As each process renders an image for a tile, that image is sent directly to the process that will display that tile. This usually results in a few processes receiving and processing the majority of the data, and is therefore usually an inefficient strategy.

ICET_STRATEGY_SPLIT Like **ICET_STRATEGY_DIRECT**, except that the tiles are split up, and each process is assigned a piece of a tile in such a way that each process receives and handles about the same amount of data. This strategy is often very efficient, but due to the large amount of messages passed, it has not proven to be very scalable or robust.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994. ISBN 0-201-63361-2.

ICET_STRATEGY_REDUCE A two phase algorithm. In the first phase, tile images are redistributed such that each process has one image for one tile. In the second phase, a "traditional" single tile composition is performed for each tile. Since each process contains an image for only one tile, all these compositions may happen simultaneously. This is a well rounded strategy that seems to perform well in a wide variety of multi-tile applications. (However, in the special case where only one tile is defined, the sequential strategy is probably better.)

ICET_STRATEGY_VTREE An extension to the binary tree algorithm for image composition. Sets up a "virtual" composition tree for each tile image. Processes that belong to multiple trees (because they render to more than one tile) are allowed to float between trees. This strategy is not quite as well load balanced as **ICET_STRATEGY_REDUCE** or **ICET_STRATEGY_SPLIT**, but has very well behaved network communication.

You can get a human-readable name using the **icetGetStrategyName** function.

```
const char *icetGetStrategyName( void );
```

The algorithms in IceT's strategies are specially designed to composite data defined on multiple tiles. Some of these algorithms, namely **ICET_STRATEGY_REDUCE** and **ICET_STRATEGY_SEQUENTIAL**, operate at least in part by compositing single images together. IceT also comes with multiple separate strategies for performing this single image compositing, and this can be selected with the **icetSingleImageStrategy** function.

```
void icetSingleImageStrategy( IceTEnum strategy );
```

IceT defines the following single image strategies that can be passed to **icetSingleImageStrategy**. These strategies are discussed in more detail in Chapter 5.

ICET_SINGLE_IMAGE_STRATEGY_AUTOMATIC Automatically chooses which single image strategy to use based on the number of processes participating in the composition.

ICET_SINGLE_IMAGE_STRATEGY_BSWAP The classic binary swap compositing algorithm. At each phase of the algorithm, each process partners with another, sends half of its image to its partner, and receives the opposite half from its partner. The processes are then partitioned into two groups that each have the same image part, and the algorithm recurses.

ICET_SINGLE_IMAGE_STRATEGY_RADIXK The radix-k acompositing algorithm is similar to binary swap except that groups of processes can be larger than two. Larger groups require more overall messages but overlap blending and communication. The size of the groups is indirectly controlled by the **ICET_MAGIC_K** environment variable or CMake variable.

ICET_SINGLE_IMAGE_STRATEGY_TREE At each phase, each process partners with another, and one of the processes sends its entire image to the other. The algorithm recurses with the group of processes that received images until only one process has an image.

By default IceT sets the single image strategy to **ICET_SINGLE_IMAGE_STRATEGY_AUTOMATIC** when a context is created. This is the single image strategy that will be used if no other is selected.

You can get a human-readable name using the **icetGetSingleImageStrategyName** function.

```
const char *icetGetSingleImageStrategyName( void );
```

Drawing Callback

Most compositing engines will simply take a group of images and combine them together. This approach, however, is unreasonable when compositing the high resolution images on a large tiled display. It is problematic for an application to create images larger than any color buffer the rendering hardware can create, and holding many of these large images can lead to a large memory profile.

Instead, the IceT algorithms deal with pieces of the overall image. The image pieces are created on demand. As such, IceT may require the same geometry to be rendered multiple times in a single frame. IceT provides the application with the most flexible way to define the rendering process: with a **drawing callback**.

A drawing callback is simply a function that your application provides IceT. When IceT needs an image, it will provide the drawing callback with the projection matrices for the section of the display being rendered. The drawing callback then returns the image rendered to those projection matrices.

IceT may call the drawing callback several times to create a single tiled image or not at all if the current bounds lie outside the current view frustum. This can have a subtle but important impact on the behavior of the drawing callback. For example, counting frames by incrementing a frame counter in the drawing callback is obviously wrong (although you could count how many times a render occurs). The drawing callback should also leave the rendering system in a state such that it will be correct for a subsequent run of the drawing callback. Any state that is assumed to be true on entrance to the drawing callback should also be true on return.

GENERIC DRAWING CALLBACK

There are two versions of drawing callbacks. The first version is a generic callback set with **icetDrawCallback**. This callback is used in conjunction with the **icetDrawFrame** function (described in the next section).

```
typedef void (*IceTDrawCallbackType)(
                              const IceTDouble *  projection_matrix,
                              const IceTDouble *  modelview_matrix,
                              const IceTFloat *   background_color,
                              const IceTInt *     readback_viewport,
                              IceTImage           result  )

void icetDrawCallback( IceTDrawCallbackType  callback  );
```

callback takes two projection matrices: *projection_matrix* and *modelview_matrix*. Each of these arguments is a 16-value array that represents a 4×4 transformation of homogeneous coordinates. The arrays store the matrices in column-major order. Thus, if the values in *projection_matrix* are $(p[0], p[1], ...p[15])$ and the values in *modelview_matrix* are $(m[0], m[1], ...m[15])$, then a vertex in object space is transformed into normalized screen coordinates by the sequence of operations

$$\begin{bmatrix} p[0] & p[4] & p[8] & p[12] \\ p[1] & p[5] & p[9] & p[13] \\ p[2] & p[6] & p[10] & p[14] \\ p[3] & p[7] & p[11] & p[15] \end{bmatrix} \begin{bmatrix} m[0] & m[4] & m[8] & m[12] \\ m[1] & m[5] & m[9] & m[13] \\ m[2] & m[6] & m[10] & m[14] \\ m[3] & m[7] & m[11] & m[15] \end{bmatrix} \begin{bmatrix} v[0] \\ v[1] \\ v[2] \\ v[3] \end{bmatrix}$$

More explicitly, if you have a point $(x, y, z, 1)$ in object space stored in a variable `object_coord`, you could transform that to world space in the callback with code like this.

```
for (row = 0; row < 4; row++) {
    world_coord[row] = 0.0;
    for (i = 0; i < 4; i++) {
        world_coord[row] += modelview_matrix[row + 4*i] * object_coord[i];
    }
}
```

Likewise, to transform the `world_coord` to normalized screen coordinates, you could apply the following code.

```
for (row = 0; row < 4; row++) {
    screen_coord[row] = 0.0;
    for (i = 0; i < 4; i++) {
        screen_coord[row] += projection_matrix[row + 4*i] * world_coord[i];
    }
}
```

If your rendering system has no need to find the world coordinates of points, you can combine the two matrices by multiplying them together like this.

```
for (row = 0; row < 4; row++) {
    for (column = 0; column < 4; column++) {
        full_matrix[row + 4*column] = 0.0;
        for (i = 0; i < 4; i++) {
            full_matrix[row + 4*column] +=
                projection_matrix[row + 4*i] * modelview_matrix[k + 4*column];
        }
    }
}
```

Normalized screen coordinates are such that everything projected onto the image has coordinates in the range $[-1, 1]$ (after dividing by the "w" homogeneous coordinate). The x and y coordinates have to be shifted to get the corresponding pixel location. The normalized screen coordinates are projected to span the physical render size (see **icetPhysicalRenderSize**), which may differ from the size of any particular tile. Also, if you are outputting depth values, IceT expects values in the range $[0, 1]$, so you will have to shift those as well. Here is a pedantic code segment to do this final transformation.

```
icetGetIntegerv(ICET_PHYSICAL_RENDER_WIDTH, &image_width);
icetGetIntegerv(ICET_PHYSICAL_RENDER_HEIGHT, &image_height);
/* Alternatively, you could get the width and height from the image passed */
/* to the callback like this.                                              */
/* image_width = icetImageGetWidth(result);                                */
/* image_height = icetImageGetHeight(result);                              */
pixel_x = (int)(image_width*0.5*(screen_coord[0]/screen_coord[3] + 1.0));
pixel_y = (int)(image_height*0.5*(screen_coord[1]/screen_coord[3] + 1.0));
depth = 0.5*(screen_coord[2]/screen_coord[3] + 1.0);
```

The drawing callback should initialize its backdrop to the *background_color*, which may be different than the background color passed to **icetDrawFrame**.

The resulting image should be rendered (or copied) into the **IceTImage**, *result*, passed to the callback. The image will be sized by the physical render width and height and its format will conform to that set by **icetSetColorFormat** and **icetSetDepthFormat**. You can get the buffers of the image with the **icetImageGetColor** and **icetImageGetDepth** functions. Data written to these buffers will become part of the image. IceT's image functions are described in more detail in the following section starting on page 43.

IceT will always send the drawing callback an image sized by the physical render viewport specified by **icetPhysicalRenderSize** for convenience. However, IceT often needs only a subset of these pixels. IceT tells the drawing callback the pixels it actually uses with the *readback_viewport* parameter. *readback_viewport* contains 4 integers specifying a region of pixels that IceT will use. The first two value specify the lower-left corner of the region and the next two specify the width and height of the region.

For example, if the *readback_viewport* is set to $(10, 15, 100, 75)$, then IceT will use only the pixels in the square between x values 10 and 109 and y values between 15 and 89 (both inclusive). All other pixels in the image will be ignored. It is not an error to provide values for the other pixels, but it is a waste of computation.

OpenGL Drawing Callback

If you are rendering with OpenGL, then you can remove many of the complexities of defining a callback by using **icetGLDrawCallback** in conjunction with **icetGLDrawFrame**.

```
typedef void (* IceTGLDrawCallbackType)( void );

void icetGLDrawCallback( IceTGLDrawCallbackType callback );
```

The OpenGL version of the drawing callback takes no arguments. It simply issues the appropriate OpenGL calls to render the geometry. IceT internally takes care of setting the appropriate transformations and clear color, and then reads back your data from the buffer specified by **icetGLSetReadBuffer** after the drawing callback returns.

The OpenGL drawing callback should *not* modify the **GL_PROJECTION_MATRIX** as this would cause IceT to place image data in the wrong location in the tiled display and improperly cull geometry. It is acceptable to add transformations to **GL_MODELVIEW_MATRIX**, but the bounding vertices given with **icetBoundingVertices** or **icetBoundingBox** (see the following section) are assumed to already be transformed by any such changes to the modelview matrix. Also, **GL_MODELVIEW_MATRIX** must be restored before the draw function returns. Therefore, any changes to **GL_MODELVIEW_MATRIX** are to be done with care and should be surrounded by a pair of glPushMatrix and glPopMatrix functions.

It is also important that the OpenGL drawing callback *not* attempt the change the clear color. In some compositing modes, IceT needs to read, modify, and change the background color. These operations will be lost if the drawing callback changes the background color, and severe color blending artifacts may result.

Specifying Geometry Bounds

IceT can nominally call the drawing callback for every tile in the display. However, in almost any real application each process has data that demonstrates some spatial locality that causes it to be projected on a relatively small section of the display. To give IceT the information it needs to prevent unnecessary renders, the application needs to provide the bounds of the local geometry. This is done using either the **icetBoundingVertices** or the **icetBoundingBox** function.

```
void icetBoundingVertices( IceTInt           size,
                           IceTEnum          type,
                           IceTSizeType      stride,
                           IceTSizeType      count,
                           const IceTVoid *  pointer  );

void icetBoundingBoxd ( IceTDouble  x_min,
                        IceTDouble  x_max,
                        IceTDouble  y_min,
                        IceTDouble  y_max,
                        IceTDouble  z_min,
                        IceTDouble  z_max  );

void icetBoundingBoxf ( IceTFloat  x_min,
                        IceTFloat  x_max,
                        IceTFloat  y_min,
                        IceTFloat  y_max,
                        IceTFloat  z_min,
                        IceTFloat  z_max  );
```

With the **icetBoundingVertices** function, you specify a set of vertices whose convex hull completely contains the geometry. The **icetBoundingBox** function is a convenience function that defines the container as an axis aligned bounding box.

Rendering

Once you have set up the IceT state as described in the previous sections of this chapter, you are ready to perform parallel rendering. Rendering is initiated in IceT by calling one of the draw frame functions.

GENERIC RENDERING

There are two frame drawing functions. The first version is independent of the rendering system and is used in conjunction with the callback set with **icetDrawCallback**. It is performed by calling **icetDrawFrame**.

```
IceTImage icetDrawFrame( const IceTDouble *  projection_matrix,
                         const IceTDouble *  modelview_matrix,
                         const IceTFloat *   background_color  );
```

Conceptually, calling **icetDrawFrame** is similar to calling the drawing callback directly. The arguments *projection_matrix*, *modelview_matrix*, and *background_color* are the same as you would potentially pass the callback (although IceT is free to change them).

The `projection_matrix` and `modelview_matrix` are 16-value arrays that represent 4×4 transformations of homogeneous coordinates. The arrays store the matrices in column-major order. Thus, if the values in `projection_matrix` are $(p[0], p[1], ...p[15])$ and the values in `modelview_matrix` are $(m[0], m[1], ...m[15])$, then a vertex in object space is transformed into normalized screen coordinates by the sequence of operations

For example, if your modelview matrix used a simple translation to move the geometry in front of the camera, you would use a matrix like this.

$$\begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The code to set the modelview matrix could look like this.

```
modelview_matrix[ 0] = 1.0;
modelview_matrix[ 1] = 0.0;
modelview_matrix[ 2] = 0.0;
modelview_matrix[ 3] = 0.0;

modelview_matrix[ 4] = 0.0;
modelview_matrix[ 5] = 1.0;
modelview_matrix[ 6] = 0.0;
modelview_matrix[ 7] = 0.0;

modelview_matrix[ 8] = 0.0;
modelview_matrix[ 9] = 0.0;
modelview_matrix[10] = 1.0;
modelview_matrix[11] = 0.0;

modelview_matrix[12] = x;
modelview_matrix[13] = y;
modelview_matrix[14] = z;
modelview_matrix[15] = 1.0;
```

As another example, consider setting the projection matrix for the perspective of a camera sitting at the origin facing down the $-z$ axis. You could use a transformation matrix like this.

$$\begin{bmatrix} \frac{f \cdot height}{width} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & -1 & -2near \\ 0 & 0 & -1 & 0 \end{bmatrix}$$
$$f = \text{cotangent}\left(\frac{fovy}{2}\right)$$

The code to set the projection matrix could look like this.

```
/* width, height: image dimensions */
/* fovy: field of view in the y direction */
/* zNear: distance from camera (at origin) to near clipping plane (at -zNear). */

f = 1.0/tan(0.5*fovy);

projection_matrix[ 0] = (f*height)/width;
projection_matrix[ 1] = 0.0;
projection_matrix[ 2] = 0.0;
projection_matrix[ 3] = 0.0;

projection_matrix[ 4] = 0.0;
projection_matrix[ 5] = f;
projection_matrix[ 6] = 0.0;
projection_matrix[ 7] = 0.0;

projection_matrix[ 8] = 0.0;
projection_matrix[ 9] = 0.0;
projection_matrix[10] = -1.0;
projection_matrix[11] = -1.0;

projection_matrix[12] = 0.0;
projection_matrix[13] = 0.0;
projection_matrix[14] = -2*zNear;
projection_matrix[15] = 0.0;
```

The *background_color* is the color in which the background should be initialized. It is the color of "empty" pixels and also the color to be blended with any transparent geometry.

icetDrawFrame returns an **IceTImage** containing the composited image displayed on this process. If the process is not displaying a tile, then the contents of the image is undefined.

OpenGL Rendering

If you are rendering with OpenGL, then you can remove many of the complexities of defining projection matrices and displaying images by using the **icetGLDrawFrame** function in conjunction with the drawing callback set with **icetGLDrawCallback**.

```
void  icetGLDrawFrame( void );
```

icetGLDrawFrame is called in basically the same way as the OpenGL drawing callback would be called directly. First, establish the OpenGL state. Setting up the **GL_PROJECTION_MATRIX** before calling **icetGLDrawFrame** is essential. It is also advisable to set up what-

ever transformations in the **GL_MODELVIEW_MATRIX** that you can before calling **icetGL-DrawFrame**. IceT will use and modify these two matrices to render regions of the tiled display. The drawing callback should behave as if neither of the matrices were modified.

By the time **icetGLDrawFrame** completes, an image will have been completely rendered and composited. If **ICET_GL_DISPLAY** is enabled, then the fully composited image is written back to the OpenGL frame buffer for display. It is the application's responsibility to synchronize the processes and swap front and back buffers. The image remaining in the frame buffer is undefined if **ICET_GL_DISPLAY** is disabled or the process is not displaying a tile.

If the OpenGL background color is set to something other than black, **ICET_GL_DISPLAY_-COLORED_BACKGROUND** should also be enabled. Displaying with **ICET_GL_DISPLAY_COL-ORED_BACKGROUND** disabled may be slightly faster (depending on graphics hardware) but can result in black rectangles in the background.

If **ICET_GL_DISPLAY_INFLATE** is enabled and the size of the renderable window (determined by the current OpenGL viewport) is greater than that of the tile being displayed, then the image will first be "inflated" to the size of the actual display. If **ICET_GL_DISPLAY_INFLATE** is disabled, the image is drawn at its original resolution at the lower left corner of the display. More details on image inflation are given in Chapter 4.

Regardless of whether it writes the fully composited image back to the display, **icetGL-DrawFrame** returns an **IceTImage** containing the composited image displayed on this process. If the process is not displaying a tile, then the contents of the image is undefined.

Image Objects

IceT uses a data type called **IceTImage** to store and pass around image data. To get image data from a generic drawing callback (described previously starting on page 36), IceT passes the callback an **IceTImage** sized to hold an image of the appropriate dimensions. Likewise, the frame drawing functions (described previously starting on page 40) return an **IceTImage**. An **IceTImage** is intended to be an opaque object that is accessed by a suite of functions provided by IceT.

IceT defines a special **null image** that can be used as a place holder when no image data is available. You can create and check for null images with the **icetImageNull** and **icetIm-ageIsNull** functions, respectively.

```
IceTImage icetImageNull( void );

IceTBoolean icetImageIsNull( IceTImage image );
```

It is good defensive programming to initialize **IceTImage** objects to null on creation. That way, all of IceT's images functions will always behave appropriately on the image, whereas the

behavior is unpredictable if the **IceTImage** is uninitialized.

```
IceTImage image = icetImageNull();
```

The functions **icetImageGetWidth**, **icetImageGetHeight**, and **icetImageGetNumPixels** return the dimensions of an image.

```
IceTSizeType  icetImageGetWidth      ( const IceTImage  image );
IceTSizeType  icetImageGetHeight     ( const IceTImage  image );
IceTSizeType  icetImageGetNumPixels  ( const IceTImage  image );
```

An **IceTImage** can hold color data, depth data, or both. Furthermore, colors and depths can be stored in different formats. The internal formats for colors and depths for an **IceTImage** can be retrieved with the **icetImageGetColorFormat** and **icetImageGetDepthFormat** functions, respectively.

```
IceTEnum  icetImageGetColorFormat ( const IceTImage  image );
IceTEnum  icetImageGetDepthFormat ( const IceTImage  image );
```

The format specifies the basic data type, the packing, and whether the color or depth data is available at all. Here is a list of possible color formats.

ICET_IMAGE_COLOR_RGBA_UBYTE	Each entry is an RGBA color tuple. Each component is valued in the range from 0 to 255 and is stored as an 8-bit integer. The buffer will always be allocated on memory boundaries such that each color value can be treated as a single 32-bit integer.
ICET_IMAGE_COLOR_RGBA_FLOAT	Each entry is an RGBA color tuple. Each component is in the range from 0.0 to 1.0 and is stored as a 32-bit float.
ICET_IMAGE_COLOR_NONE	No color values are stored in the image.

Here is a list of possible depth formats.

ICET_IMAGE_DEPTH_FLOAT	Each entry is in the range from 0.0 (near plane) to 1.0 (far plane) and is stored as a 32-bit float.
ICET_IMAGE_DEPTH_NONE	No depth values are stored in the image.

An **IceTImage** stores the color and depth data in separate buffers. You can use the **icetImageGetColor** and **icetImageGetDepth** functions to retrieve pointers to this data. A drawing callback must use these functions to get buffers to write data into.

```
IceTUByte *  icetImageGetColorub  ( IceTImage  image  );
IceTUInt *   icetImageGetColorui  ( IceTImage  image  );
IceTFloat *  icetImageGetColorf   ( IceTImage  image  );

IceTFloat *  icetImageGetDepthf   ( IceTImage  image  );

const IceTUByte *  icetImageGetColorcub  ( const IceTImage  image  );
const IceTUInt *   icetImageGetColorcui  ( const IceTImage  image  );
const IceTFloat *  icetImageGetColorcf   ( const IceTImage  image  );

const IceTFloat *  icetImageGetDepthcf   ( const IceTImage  image  );
```

The various forms of **icetImageGetColor** and **icetImageGetDepth** return typed arrays for the buffer of data. The type of the data must conform to the internal format of the data; the functions will raise an error otherwise.

If you want to use image data of a specific format, you can use one of the **icetImageCopyColor** or **icetImageCopyDepth** functions. With these functions, you give an allocated array and a specific color format, and the data for that array will be copied into your buffer in the desired format.

```
void  icetImageCopyColorub ( const IceTImage  image,
                             IceTUByte *      color_buffer,
                             IceTEnum         color_format  );

void  icetImageCopyColorf  ( const IceTImage  image,
                             IceTFloat *      color_buffer,
                             IceTEnum         color_format  );

void  icetImageCopyDepthf  ( const IceTImage  image,
                             IceTFloat *      depth_buffer,
                             IceTEnum         depth_format  );
```

Chapter 4

Customizing Compositing

If you have been reading this document from the beginning, then you already know enough to use IceT for many typical rendering applications. Chapters 2 and 3 describe how to build and link IceT, establish an IceT context in your application, and to leverage IceT to make your rendering parallel. This chapter describes the many features IceT provides to let you customize the image compositing to your application.

Compositing Operation

IceT is classified as a **sort-last** type of parallel rendering library, as discussed in Chapter 1. Basically, this means that each process renders images independently, and then these images, each comprising a different partition of the geometry, are combined together in a process called **compositing**.

To combine two images together, a **compositing operation** is applied to every corresponding pair of pixels. Three or more images are combined by applying the compositing operation multiple times to eventually reduce everything to one image. (The compositing operations supported by IceT are associative, so the order is flexible. IceT takes advantage of this fact to efficiently perform the compositing in parallel.)

IceT supports two compositing operations, set with **icetCompositeMode**.

```
void icetCompositeMode( IceTEnum mode );
```

The first type of compositing operation, **ICET_COMPOSITE_MODE_Z_BUFFER**, is a depth comparison and the other, **ICET_COMPOSITE_MODE_BLEND**, is an alpha blend. The depth comparison is a bit faster and is easier to use, but only works for opaque surfaces. If you are performing **volume rendering**, the translucent rendering of 3-dimensional volumes, or any other rendering that involves transparent data, then you will have to use the alpha blend compositing operation.

Each compositing operation relies on certain buffers to exist (or not exist) in images. For example, z-buffer compositing can use a color buffer and requires a depth buffer whereas blended compositing requires a color buffer and cannot work with a depth buffer. The buffers created in images by IceT, and their formats, is controlled by the **icetSetColorFormat** and **icetSet-**

DepthFormat functions. It is important to ensure that the setting for **icetCompositeMode** is compatible with the settings for **icetSetColorFormat** and **icetSetDepthFormat**.

```
void icetSetColorFormat ( IceTEnum color_format );
void icetSetDepthFormat ( IceTEnum depth_format );
```

The following *color_format*s are valid for use in **icetSetColorFormat**.

ICET_IMAGE_COLOR_RGBA_UBYTE	Each entry is an RGBA color tuple. Each component is valued in the range from 0 to 255 and is stored as an 8-bit integer. The buffer will always be allocated on memory boundaries such that each color value can be treated as a single 32-bit integer.
ICET_IMAGE_COLOR_RGBA_FLOAT	Each entry is an RGBA color tuple. Each component is in the range from 0.0 to 1.0 and is stored as a 32-bit float.
ICET_IMAGE_COLOR_NONE	No color values are stored in the image.

The following *depth_format*s are valid for use in **icetSetDepthFormat**.

ICET_IMAGE_DEPTH_FLOAT	Each entry is in the range from 0.0 (near plane) to 1.0 (far plane) and is stored as a 32-bit float.
ICET_IMAGE_DEPTH_NONE	No depth values are stored in the image.

Z-BUFFER COMPOSITING

Z-buffer compositing takes advantage of the same hidden surface removal already taking place in the OpenGL pipeline. IceT pulls the z-buffer (also often known as the **depth buffer**) from the OpenGL image buffers. The compositing operation then just compares the depth values of two pixels and chooses the one that is closer.

Z-buffer compositing is used when the composite mode (set with **icetCompositeMode**) is **ICET_COMPOSITE_MODE_Z_BUFFER**. Z-buffer compositing also requires a depth buffer. An error will occur during compositing if z-buffer compositing is being used without a depth buffer (i.e. **icetSetDepthFormat** is set to **ICET_IMAGE_DEPTH_NONE**.)

By default, z-buffer compositing is enabled and both the the color and the depth buffer are selected as input buffers. Also by default IceT will *not* produce a depth buffer. (Not computing the depth buffer may save some network transfer time.) This behavior is controlled by the **ICET_COMPOSITE_ONE_BUFFER** option, which is enabled by default.

If you need the depth buffer composited in addition to the color buffer (for example, to help with a picking operation), you can do so by simply disabling the **ICET_COMPOSITE_ONE_BUFFER** option.

```
icetCompositeMode(ICET_COMPOSITE_MODE_Z_BUFFER);
icetSetColorFormat(ICET_IMAGE_COLOR_RGBA_UBYTE);
icetSetDepthFormat(ICET_IMAGE_DEPTH_FLOAT);
icetDisable(ICET_COMPOSITE_ONE_BUFFER);
```

Alternatively, if you only need the depth buffer (for example, as a shadow map), you can do so by setting the color format to **ICET_IMAGE_COLOR_NONE**. In this case, the **ICET_COMPOSITE_ONE_BUFFER** option will have no effect

```
icetCompositeMode(ICET_COMPOSITE_MODE_Z_BUFFER);
icetSetColorFormat(ICET_IMAGE_COLOR_NONE);
icetSetDepthFormat(ICET_IMAGE_DEPTH_FLOAT);
```

VOLUME RENDERING (AND OTHER TRANSPARENT OBJECTS)

A well known limitation to z-buffer compositing — and the z-buffer hidden surface removal algorithm in general — is that it only works with opaque objects. You will get invalid results if you try to apply z-buffer compositing on transparent objects.

There are two fundamental problems with the z-buffer compositing operation when dealing with translucent pixels. The first problem is that you cannot simply pick the nearest color value. You must **blend** the front pixel's color with the back pixel's color. The second problem is that the color blending is order dependent. That is, you have to know which pixels are in front of others. Although it is technically possible to use z-buffer values to determine the ordering of a pair of pixels, making sure that all the pixels get composited in the correct order requires additional information about and constraints on the geometry.

When z-buffer compositing is not applicable, you must use **blended compositing**. To use blended compositing, set the composite mode (with **icetCompositeMode**) to **ICET_COMPOSITE_MODE_BLEND** and turn off depth buffers (i.e. **icetSetDepthFormat** is set to **ICET_IMAGE_DEPTH_NONE**).

```
icetCompositeMode(ICET_COMPOSITE_MODE_BLEND);
icetSetColorFormat(ICET_IMAGE_COLOR_RGBA_UBYTE);
icetSetDepthFormat(ICET_IMAGE_DEPTH_NONE);
```

The blending composite operator relies on the **alpha** (α) channel of the color buffer (the A in RGBA colors). Note that when using OpenGL, the alpha values must actually be available in

the OpenGL color buffers in order for blended compositing to work. Many applications create OpenGL buffers without alpha bit planes in them because they are often not necessary to render images in serial. Make sure your application creates alpha bit planes before attempting to composite translucent images with IceT (or any other library).

The blending operation is the standard **over/under operator** defined in the seminal 1984 Porter and Duff paper.

$$C_o \leftarrow C_f + C_b(1 - \alpha_f) \tag{4.1}$$

where C is an RGBA color vector, α is the alpha component of a color vector, and the f, b, and o subscripts denote the front, back, and output values, respectively.

Each color in Equation 4.1 represents a **pre-multiplied color**, meaning that the red, green, and blue values are scaled by the alpha parameter. Thus, a fully red color at half transparency is represented by the vector $\langle 0.5, 0, 0, 0.5 \rangle$ rather than $\langle 1, 0, 0, 0.5 \rangle$. In pre-multiplied colors, none of the red, green, or blue values ever exceed the alpha value. Note that colors are often provided in OpenGL as non-pre-multiplied values, and the blending equation $C_o \leftarrow C_f \alpha_f + C_b(1 - \alpha_f)$ is used instead of the one in Equation 4.1. Although this blending gives the correct RGB color, it computes an invalid alpha parameter, so watch out!

Simply turning on blended compositing is not sufficient to render translucent objects. You must also tell IceT to perform **ordered compositing**. In ordered compositing, you must have a **visibility ordering**. Given any two processes, a visibility ordering ensures and determines that all of the geometry in one process is in front of or behind all the geometry in each of the other process with respect to the camera. In some cases, such as when volume rendering a 3D Cartesian grid of points distributed in blocks to processes, finding the visibility ordering is straightforward. In other cases, such as when rendering unstructured collections of polygons or polyhedra, it can be difficult to ensure that a visibility ordering exists and can be found. Doing so may be the most challenging part of creating a parallel rendering application. An example of creating a visibility ordering from unstructured data can be found in the ParaView application, and the implementation is detailed in the following paper:

> Kenneth Moreland, Lisa Avila, and Lee Ann Fisk. "Parallel Unstructured Volume Rendering in ParaView," In *Visualization and Data Analysis 2007, Proceedings of SPIE-IS&T Electronic Imaging*, January 2007, pp. 64950F-1–12.

Ordered compositing is turned on by simply passing the **`ICET_ORDERED_COMPOSITE`** flag to **`icetEnable`**.

```
icetEnable(ICET_ORDERED_COMPOSITE);
```

Once ordered compositing is enabled, it is very important to use **`icetCompositeOrder`** to specify the visibility order of the geometry associated with each process. This must generally be done before each call to **`icetDrawFrame`** or **`icetGLDrawFrame`**.

```
void icetCompositeOrder( const IceTInt * process_ranks );
```

The **icetCompositeOrder** function takes an array of processes. It is assumed that the geometry of the first process in the list is in front of the rest of the processes; the geometry of the second process in the list is in front of all the processes except the first, and so on. The visibility order often changes when the camera angle changes, so it is important to recompute and report a new composite order on every frame.

Be aware that not all strategies support ordered compositing. If the current strategy does not support ordered compositing, then the **ICET_ORDERED_COMPOSITE** flag is ignored. Consult the documentation in Chapter 5 or the documentation for the **icetStrategy** command to determine which strategies support ordered compositing. In any case, you can check the **ICET_STRATEGY_SUPPORTS_ORDERING** state variable to determine if the current compositing strategy supports ordered compositing.

One final thing to worry about when using blended compositing is to make sure that the background color does not interfere with the compositing. Because the visibility order is important, you need to make sure that none of the processes render with a background (except perhaps the process nearest the rear). For example, let us say you want to render an image with a blue background. Let us also say that process A's geometry is in front of process B's geometry. Process A cannot render its geometry on top of a blue background because that background should really also be behind the geometry of process B, and the resulting image will be invalid.

If your background is a solid color, then IceT can fix this problem automatically. Both **icetDrawFrame** and **icetGLDrawFrame** have the ability take a solid background color and modify it as appropriate for compositing. **icetDrawFrame** takes the background color as an explicit parameter whereas **icetGLDrawFrame** implicitly gets the background color from the OpenGL clear color.

When the **ICET_CORRECT_COLORED_BACKGROUND** feature is enabled and blended compositing is on, IceT will change the background to $\langle 0,0,0,0 \rangle$, perform the rendering and compositing, and blend the result into the specified background color.

If you are using **icetGLDrawFrame** to render with the OpenGL layer and if you do not actually need to use the image returned from **icetGLDrawFrame**, you can use the **ICET_GL_DISPLAY_COLORED_BACKGROUND** option instead.

ICET_GL_DISPLAY_COLORED_BACKGROUND operates similar to **ICET_CORRECT_COLORED_BACKGROUND** with the exception that it uses the OpenGL graphics hardware to blend the composited image to the colored background, and may therefore get a modest performance increase. However, it also means that the result will not be available in the memory buffer returned by **icetGLDrawFrame**.

Image Inflation

Because IceT is an image-based sort-last parallel rendering library, its overhead is proportional to the size of the images being generated. Thus, large displays can limit the maximum rendering frame rate that can be achieved.

A simple way to increase the frame rate is to reduce the resolution of the images being displayed. If the display resolution is larger than necessary (and "larger than necessary" is a flexible metric that can change regularly as an application runs), then you can render smaller images and then **inflate** the images to fill the display. A major use case for a reduced resolution image is for maintaining application interactivity. Many applications, particularly visualization applications, contain bursts of interactivity. The user will interact with the data (move the camera or objects) and then hold still and analyze the results. While interacting, application responsiveness is much more important than image details, so during this time a lower resolution image can be rendered and inflated. When the user stops interacting and starts analyzing, a full resolution image can be created.

You can instruct IceT to render and composite smaller images by simply specifying a lower resolution display with the **icetAddTile** function. If you are frequently switching the resolution of the images being generated (which is common), then you can use IceT state management to switch states. First, use **icetCreateContext** and **icetCopyState** to create a duplicate state. Then change the display of one of the states to a lower resolution with **icetAddTile**. As the application runs, use **icetSetContext** to swap between the different resolutions. See Chapter 3 for details on using these functions.

Between rendering and display, the smaller images must be inflated to fill the display. An application can always perform this inflation itself (and that is probably necessary if the images are shipped to a remote display). When using IceT's OpenGL layer (i.e. rendering with calls to **icetGLDrawFrame**) and IceT is displaying the data (i.e. **ICET_GL_DISPLAY** is enabled), IceT has the ability to automatically inflate the images. Turn on this feature by enabling **ICET_GL_DISPLAY_INFLATE**. IceT contains two modes for inflating images: using the CPU or using texture mapping in OpenGL. When **ICET_GL_DISPLAY_INFLATE_WITH_HARDWARE** is enabled (the default), then texture mapping is used. In either case, **icetGLDrawFrame** returns the smaller image size specified by **icetAddTile**.

One final note: Regardless of what size you set for the displays in **icetAddTile**, you should render images as large as possible (by setting **icetPhysicalRenderSize** or **glViewport** as large as possible). The size of the rendered images and the size of the tile images can be different so long as each rendered image is at least as large as the largest tile image. In fact, it is advantageous to have the rendered images larger than the specified tiles. The first reason is that the **ICET_GL_DISPLAY_INFLATE** feature fills the image to the OpenGL viewport. If the dimensions the two are the same, then no inflation will actually take place. The second reason is that IceT will use the entire renderable space. For a multi-tile display, this can dramatically reduce the number of times the render callback needs to be called. Thus, in general it is best to keep the rendered images as large as possible.

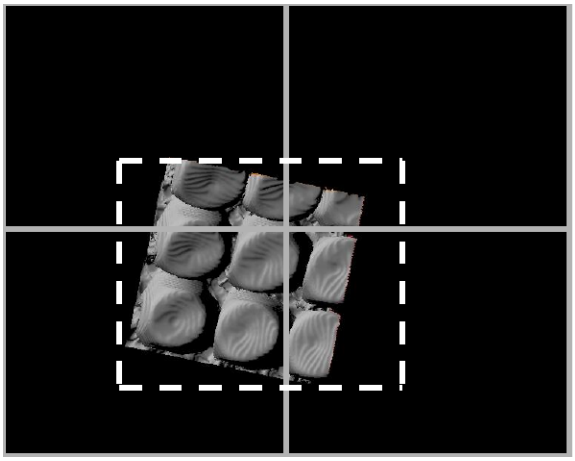

Figure 4.1. Even though geometry may straddle tile boundaries, we may be able to render it all in one pass by "floating" the viewport.

Floating Viewport

Consider the geometry shown in Figure 4.1 that projects onto a screen space that fits within a single tile but is moved in the horizontal and vertical directions so that it straddles four tiles. If the system limits itself to projecting onto physical tiles, the processor must render four images even though it could generate a single image that contains the entire geometry with the exact same pixel spacing. Instead of rendering four tiles, IceT can **float** the viewport in the global display to the space straddling the tiles. That is, IceT may project the geometry to the space shown by the dotted line in Figure 4.1 and split the resulting image back into pieces that can be displayed directly on each tile. Hence, the system does not need to render any polygon more than once.

When a processor's geometry fits within the floating viewport, it can cut the rendering time dramatically. This is most likely to happen when the number of tiles is small compared to the number of processors and the spatial coherency of the data is good.

The floating viewport is always enabled by default. You can disable it by calling **`icetDisable`** with the **`ICET_FLOATING_VIEWPORT`** identifier. In general, there is not much reason to turn off the floating viewport. The only real reason to turn off the floating viewport is to prevent IceT from changing the perspective matrix when in single tile mode. However, IceT will change the perspective matrix anyway when rendering with more than one tile, so any application that might render to a tiled display should simply leave the floating viewport option on.

Active-Pixel Encoding

Because each processor renders only a fraction of the total geometry, the geometry often occupies only a fraction of the screen space in some or all of the tiles in which it lies. Consequently, the initial images distributed between processors at the beginning of composition often have a significant amount of blank space within them. Explicitly sending this data between processors is a waste of bandwidth. Transferring sparse image data rather than full image data is a well-known way to reduce network overhead. So far, our best method to do this has been with active-pixel encoding.

Active-pixel encoding is a form of run-length encoding. A traditional run-length encoding groups pixels into contiguous groups where the color or depth does not change. However, in a practical 3D rendering, both the color and depth change almost everywhere except in the background areas where nothing is rendered. To take advantage of this, images are grouped into alternating run lengths of **active pixels**, pixels that contain geometry information, and **inactive pixels**, pixels that have no geometry drawn on them. The active-pixel run length is followed by pairs of color and depth values (or just one of the two if that is the only data available). The inactive pixels are not accompanied by any color or depth information. The depth information is assumed to be of maximum depth, and the color values are ignored since they contain no geometry information.

There are many other ways to encode sparse images and reduce data redundancy. However, we are particularly enamored with our active-pixel encoding for this application because it exhibits all of the following properties:

Fast encoding Image encoding requires each pixel to be visited exactly once. Each visit includes a single alpha or depth comparison, a single addition, and at most one copy.

Free decoding Processors typically perform a compositing as soon as they receive incoming data. The compositing can be done directly against an image that is still encoded in sparse form. In fact, the compositing can skip the comparisons for the inactive pixels. Thus doing compositing against encoded images is often faster than against unencoded images.

Effective compression During the early stages of composition when the most image data must be transferred, the sparse data is commonly less than one fifth the size of the original data.

Good worst case behavior No image will ever grow by more than a few bytes of header information. Images that have geometry drawn on every pixel will only have one run length. Even images that alternate between active and inactive status for every pixel, and hence have a run length for every pixel, do not grow when encoded. The number of bytes required to record two run lengths, which are stored as 16-bit integers each, is no more than the number of bytes saved by not recording either color or depth data for a single inactive pixel, which is at least 32-bits. Thus, there is no penalty for recording run lengths of size one.

Active-pixel encoding is performed automatically during the compositing process. There is currently no way to turn it off.

Interlaced Images

Although active pixel encoding almost always improves the performance of compositing, it does introduce an issue of load balancing. As images are partitioned, some regions will have more active pixels than others. By balancing the active pixels assigned to regions, the parallel compositing becomes better load balanced and performance can improve even further.

The most straightforward way of balancing active pixels is to **interlace** the images. An image is interlaced by rearranging pixels in a different order. This reordering of pixels is designed such that when the images are later partitioned, each partition gets pixels from all over the images. Consequently, regions with many active pixels are distributed to all the partitions.

Although image interlacing can provide a significant performance increase, it also incurs an overhead caused by shuffling pixels around. This overhead can potentially happen in two places during parallel compositing. The first overhead is the shuffling of pixels before any compositing or message transfers take place. This overhead tends to be low because it occurs when images are their most sparse and the work is distributed amongst all the processes. The second overhead is the reshuffling after compositing completes to restore the proper order of the pixels. This second shuffling is substantial as it happens when images are at their most full and the maximum amount of pixels must be copied. Furthermore, because pixels must be shuffled over the entire image, this reshuffling must happen after image data is collected on a single process. Thus, the reshuffling happens on a single process while all others remain idle.

Classical implementation of image interlacing shuffle pixels in regions, but the regions chosen are usually arbitrary (scanlines is a common region to pick). However, the image interlacing algorithm in IceT chooses regions that completely avoid the need for the second, and most time consuming, pixel reshuffling. The algorithm is based on the simple observation that at the completion of many parallel compositing algorithms, the final image is partitioned and distributed among all the processes in contiguous pieces. If we arrange our initial shuffling such that each of the partitions remain a contiguous set of pixels, then we do not need the final reshuffling at all.

IceT's minimal-copy image interlacing is demonstrated in Figure 4.2. Rather than picking arbitrary partitions, such as scan lines, to interlace, IceT's interlacing uses the partitions that the composite will create anyway. The image with the interlaced block is composited as normal. Each resulting image partition is already intact, it is only the implicit offsets that need to be adjusted.

Image interlacing is always on by default. It can be turned off by calling **`icetDisable`** with **`ICET_INTERLACE_IMAGES`**. Image interlacing is only a hint, and compositing strategies are not obligated to follow it. In any case, the observable behavior between interlaced and non-interlaced images is the same except for in compositing times. In situations where each process renders to a small portion of an image, the overhead for image interlacing is low but its benefits are high. However, in cases where every process renders geometry over the entire view (indicative of a poor distribution of geometry), then the overhead for image interlacing becomes higher but the benefit becomes lower.

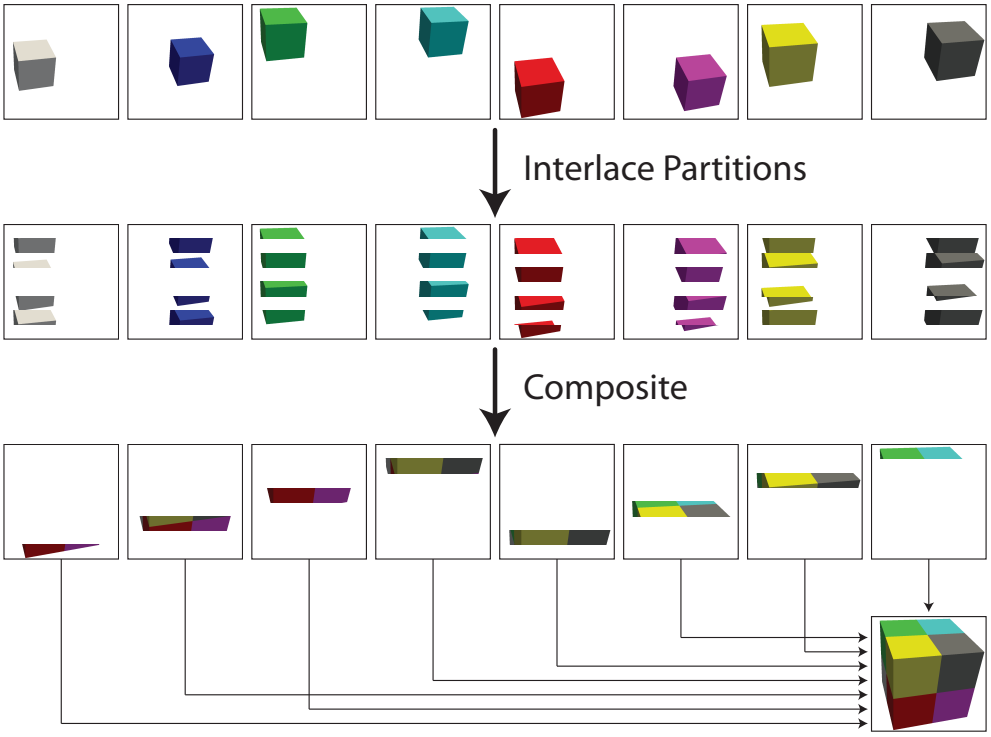

Figure 4.2. Pixel shuffling in IceT's image interlacing.

Data Replication

The primary advantage of IceT's parallel rendering algorithms, and sort-last rendering algorithms in general, is that they are very scalable with respect to the size of the input geometry. That is, there is no overhead to adding more geometry other than the time it takes hardware to render and there is only a logarithmic overhead for adding processors to the job.

The down side of a sort-last approach is that the image compositing overhead must be incurred regardless of how little geometry is being rendered. This overhead limits the maximum frame rate that can be achieved by the parallel rendering. Consequently, the parallel rendering can potentially be slower than the serial rendering if the amount of geometry being rendered is small.

One possible way to get higher frame rates with smaller geometries would be to switch to a different parallel rendering mode, but doing so is unnecessarily complicated. Another possibility is to collect the data on a single process and circumvent the IceT library entirely. This approach is fine when using single tile mode where the image is displayed at a single location, but is not at all straightforward when displaying to a tiled display.

IceT provides a better solution than either of the previous two approaches. If the image compositing work is dominating the rendering time, you can set up a **data replication group**. To set up a data replication group, you partition the geometry using fewer partitions than processes, and

then you share each partition with multiple processes. The processes that share a data partition are a replication group. IceT will divide the compositing work for each replication group amongst the processes in the group. In essence, you are adding geometry rendering work to remove image compositing work.

One of the most common uses for data replication groups is to simply replicate the same geometry on all processes. This is helpful, for example, if your application supports lower levels of detail for interaction. The lower level of detail can be replicated on all processes. However, you could also conceivably arrange for any amount of replication between all replicated and none replicated for a consistently appropriate overhead as the amount of data grows.

To set up data replication groups, it is your responsibility to partition and replicate geometry. (IceT knows nothing about geometry.) You then report what the data replication groups are with the **icetDataReplicationGroup** function.

```
void icetDataReplicationGroup( IceTInt          size,
                               const IceTInt *  processes  );
```

icetDataReplicationGroup simply takes an array defining the replication group that the local process belongs to. It is important to ensure that all processes belonging to a group provide the same array to **icetDataReplicationGroup**. As a convenience, IceT also provides the **icetDataReplicationGroupColor** function that allows you to define the data replication groups by assigning an identifier (i.e. color) to each partition and having each process report the partition color in which it belongs.

```
void icetDataReplicationGroupColor( IceTInt  color  );
```

As an example, let us say that processes 0–3 share the same geometry, 4–7 share the same geometry, 8–11 share the same geometry, and so on. These replication groups could be reported with the following call (where `rank` is the local process id as stored in the **ICET_RANK** state variable).

```
icetDataReplicationGroupColor(rank/4);
```

The data replication group is stored in the **ICET_DATA_REPLICATION_GROUP** state variable (retrievable with **icetGet**). The length of the group array is stored in **ICET_DATA_REPLICATION_GROUP_SIZE**. The data replication group array is available regardless of whether you used **icetDataReplicationGroup** or **icetDataReplicationGroupColor** to define the group. The default value is an array with one value: the local process.

Compositing Network Hints

By its nature, image compositing requires a significant amount of communication amongst processes in a parallel computer. Each compositing algorithm contained in IceT follows a particular

pattern of communication, referred to as its **compositing network**. These algorithms and their compositing networks are described in Chapter 5.

Most compositing networks are fixed although a few have some possible variability. Any variations in the compositing networks are automatically chosen, but it is possible to provide hints. Because the relative efficiency of different compositing networks is effected more by the underlying hardware than in the application and data running it, these hints are specified by either build options or by environment variables.

Build options are specified as CMake variables. These CMake variables are set using the CMake program at the beginning of the build as described in Chapter 2. These variables are listed as "advanced," so you will need to turn on advanced variables in the CMake to see them.

For each one of these CMake build variables, IceT also recognizes a corresponding environment variable with the same name at run time. If the environment variable is defined, it will be used in place of the option set at run time. The environment variables are read when the IceT context is first created, so subsequent changes to the environment variables will have no effect.

The following CMake and environment variable hints are available.

`ICET_MAGIC_K` The radix-k single image strategy described in Chapter 5 starting on page 66 operates by communicating within groups of processes of size k. IceT's implementation of radix-k automatically determines k, but does so by selecting values that are as close as possible to a **magic k** value. This magic k value is set with the **`ICET_MAGIC_K`** CMake build variable or environment variable.

`ICET_MAX_IMAGE_SPLIT` The parallel image compositing algorithms in IceT maintain load balancing by dividing images amongst processes to allow the processes to concurrently composite pixels in independent partitions of the image. Typically, IceT subsequently collects the images to the display nodes. This collection can be one of the longest operations during the compositing, particularly when there are many processes. It is possible to speed up the collection at the expense of the compositing by limiting the maximum number of times an image may be split. This limit is set with the **`ICET_MAX_IMAGE_SPLIT`** CMake build variable or environment variable. Be advised that this option is only a suggestion to compositing. It is still possible for images to be split beyond this limit. IceT will still behave correctly either way.

These values may be queried (but not set) at runtime using **`icetGet`** with a state variable identifier of the same name.

Image Partition Collection

Many parallel compositing algorithms, including several in IceT, function by partitioning images and distributing the pieces. These algorithms, described in more detail in Chapter 5, even-

tually have a composited image partitioned and distributed among most or all of the processes involved in the operation.

For most practical purposes, such as displaying the images, the image partitions must be collected, and by default IceT collects each image to its associated display process. However, there exist some use cases where this collection may not be necessary. For example, if the images are only to be written to a parallel file system, then it may be as efficient or more efficient to collectively write partitions from multiple processes.

The collection of image data to the display process may take a significant portion of the overall compositing time, but it is usually necessary. When it is not necessary, it can be disabled by calling **icetDisable** with **ICET_COLLECT_IMAGES**. When this option is turned off, the strategy has the option of leaving images partitioned among processes. Each process containing part of a tile's image will return the entire buffer from **icetDrawFrame** or **icetGLDrawFrame** in an **IceTImage** object. However, only certain pixels will be valid. The state variables **ICET_VALID_PIXELS_TILE**, **ICET_VALID_PIXELS_OFFSET**, and **ICET_VALID_PIXELS_NUM** give which tile the pixels belong to and what range of pixels are valid.

The **ICET_VALID_PIXELS_TILE** state variable gives the tile for which the last image returned from **icetDrawFrame** or **icetGLDrawFrame** contains pixels. Each process has its own value. If the last call to **icetDrawFrame** or **icetGLDrawFrame** did not return pixels for the local process, then this state variable is set to -1.

The **ICET_VALID_PIXELS_OFFSET** and **ICET_VALID_PIXELS_NUM** give the range of valid pixels for the last image returned from **icetDrawFrame** or **icetGLDrawFrame**. Given the arrays of pixels returned with the **icetImageGetColor** and **icetImageGetDepth** functions, the valid pixels start at the pixel indexed by **ICET_VALID_PIXELS_OFFSET** and continue for **ICET_VALID_PIXELS_NUM**. The tile to which these pixels belong are captured in the **ICET_VALID_PIXELS_TILE** state variable. If the last call to **icetDrawFrame** or **icetGLDrawFrame** did not return pixels for the local process, **ICET_VALID_PIXELS_NUM** is set to 0.

The **ICET_VALID_PIXELS_TILE**, **ICET_VALID_PIXELS_OFFSET**, and **ICET_VALID_PIXELS_NUM** are only really useful when **ICET_COLLECT_IMAGES** is disabled. When **ICET_COLLECT_IMAGES** is enabled, it is always the case that each display process has the entire image (**ICET_VALID_PIXELS_TILE** set to the tile, **ICET_VALID_PIXELS_OFFSET** set to 0 and **ICET_VALID_PIXELS_NUM** set to the total image size), and it is always the case that each process not displaying a tile has no image (**ICET_VALID_PIXELS_TILE** set to -1 and **ICET_VALID_PIXELS_NUM** set to 0).

Timing (and Other Metrics)

Whenever **icetDrawFrame** or **icetGLDrawFrame** is called, IceT measures the amount of time spent in the various tasks required for parallel rendering. These timings are stored in the

IceT state and can be retrieved with **icetGet**. The state variables containing these timing metrics (in seconds) are as follows.

ICET_RENDER_TIME The total time spent in the drawing callback during the last call to **icetDrawFrame** or **icetGLDrawFrame**.

ICET_BUFFER_READ_TIME The total time spent copying buffer data and reading from OpenGL buffers during the last call to **icetDrawFrame** or **icetGLDrawFrame**.

ICET_BUFFER_WRITE_TIME The total time spent writing to OpenGL buffers during the last call to **icetGLDrawFrame**. Always set to 0.0 after a call to **icetDrawFrame**.

ICET_COMPRESS_TIME The total time spent in compressing image data using active pixel encoding during the last call to **icetDrawFrame** or **icetGLDrawFrame**.

ICET_BLEND_TIME /ICET_COMPARE_TIME The total time spent in performing Z comparisons or color blending of images during the last call to **icetDrawFrame** or **icetGLDrawFrame**. These two variables always return the same value.

ICET_COLLECT_TIME The total time spent in collecting image fragments to display processes during the last call to **icetDrawFrame** or **icetGLDrawFrame**.

ICET_TOTAL_DRAW_TIME The total time spent in the last call to **icetDrawFrame** or **icetGLDrawFrame**. This includes all the time to render, read back, compress, and composite images.

ICET_COMPOSITE_TIME The total time spent in compositing during the last call to **icetDrawFrame** or **icetGLDrawFrame**. Equal to **ICET_TOTAL_DRAW_TIME** − **ICET_RENDER_TIME** − **ICET_BUFFER_READ_TIME** − **ICET_BUFFER_WRITE_TIME**.

In addition to timing how long rendering and compositing takes, IceT also keeps track of how much data is transmitted during compositing. The state variable **ICET_BYTES_SENT** stores the total number of bytes sent by the calling process for transferring image data during the last call to **icetDrawFrame**. Obviously, each process could have a different value for **ICET_BYTES_SENT**.

IceT also keeps track of the number of times **icetDrawFrame** or **icetGLDrawFrame** has been called. This number is stored in **ICET_FRAME_COUNT**.

Chapter 5

Strategies

IceT contains several parallel image compositing algorithms. The type of compositing algorithm to use is selected by choosing a **strategy**. This chapter describes the underlying algorithm of each strategy. This user's guide will give qualitative comparisons between the strategies, but for a more quantitative analysis, see the following paper.

> Kenneth Moreland, Brian Wylie, and Constantine Pavlakos. "Sort-last parallel rendering for viewing extremely large data sets on tile displays," In *Proceedings of IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, October 2001, pp. 85–154.

A strategy is specified using the **icetStrategy** function.

```
void icetStrategy( IceTEnum strategy );
```

The *strategy* is set to one of the following identifiers.

ICET_STRATEGY_SEQUENTIAL Basically applies a "traditional" single tile composition (such as binary swap) to each tile in the order they were defined. Because each process must take part in the composition of each tile regardless of whether they draw into it, this strategy is usually inefficient when compositing for more than one tile, but is recommended for the single tile case because it bypasses some of the communication necessary for the other multi-tile strategies.

ICET_STRATEGY_DIRECT As each process renders an image for a tile, that image is sent directly to the process that will display that tile. This usually results in a few processes receiving and processing the majority of the data, and is therefore usually an inefficient strategy.

ICET_STRATEGY_SPLIT Like **ICET_STRATEGY_DIRECT**, except that the tiles are split up, and each process is assigned a piece of a tile in such a way that each process receives and handles about the same amount of data. This strategy is often very efficient, but due to the large amount of messages passed, it has not proven to be very scalable or robust.

ICET_STRATEGY_REDUCE A two phase algorithm. In the first phase, tile images are redistributed such that each process has one image for one tile. In the second phase, a "traditional" single tile composition is performed for each tile. Since each process contains an image for only one tile, all these compositions may happen simultaneously. This is a well rounded strategy that seems to perform well in a wide variety of multi-tile applications. (However, in the special case where only one tile is defined, the sequential strategy is probably better.)

ICET_STRATEGY_VTREE An extension to the binary tree algorithm for image composition. Sets up a "virtual" composition tree for each tile image. Processes that belong to multiple trees (because they render to more than one tile) are allowed to float between trees. This strategy is not quite as well load balanced as **ICET_STRATEGY_REDUCE** or **ICET_STRATEGY_SPLIT**, but has very well behaved network communication.

A string documenting the current strategy can be retrieved with the **icetGetStrategyName** function. The following sections describe the strategies in more detail.

To describe the IceT compositing algorithms, we will use the example parallel rendering problem shown in Figure 5.1 where 6 processes are each rendering their own piece of a shuttle model to a two tile display.

In this example, processes are denoted, in no particular order, by the colors gray, red, blue, green, purple, and orange. The colors of the geometry correspond to the process that generated each piece. Image boarder colors denote the process that generates and holds that image. (Apologies to those having troubles resolving the colors due to poor display, printout, or vision deficiencies. It should not be hard to follow the descriptions either way.)

Single Image Compositing

Before discussing the multi-tile image compositing algorithms implemented by IceT, we visit the standard single image compositing algorithms. You cannot directly use a single image compositing algorithm as a strategy (most of the multi-tile algorithms work well in "single-tile" mode), but these compositing algorithms are used as "subroutines" in some of the multi-tile algorithms. A reference to a **single image composite network** in the subsequent compositing algorithm descriptions refers to the algorithms described here.

You can, however, choose which single image strategy is used by the main multi-tile strategy. This is selected with the **icetSingleImageStrategy** function.

```
void icetSingleImageStrategy( IceTEnum strategy );
```

The *strategy* is set to one of the following enumerations.

ICET_SINGLE_IMAGE_STRATEGY_AUTOMATIC Automatically chooses which single image strategy to use based on the number of processes participating in the composition.

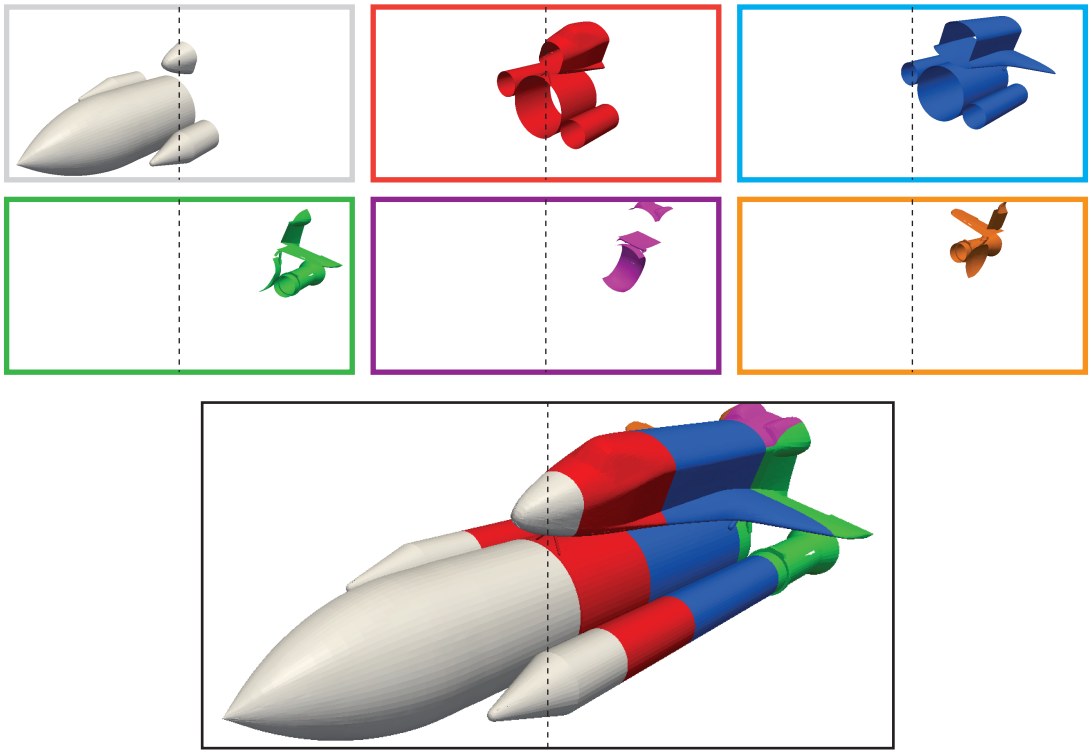

Figure 5.1. An example of six processes rendering to two tiles (top) and their composited image (bottom).

ICET_SINGLE_IMAGE_STRATEGY_BSWAP The classic binary swap compositing algorithm. At each phase of the algorithm, each process partners with another, sends half of its image to its partner, and receives the opposite half from its partner. The processes are then partitioned into two groups that each have the same image part, and the algorithm recurses.

ICET_SINGLE_IMAGE_STRATEGY_RADIXK The radix-k acompositing algorithm is similar to binary swap except that groups of processes can be larger than two. Larger groups require more overall messages but overlap blending and communication. The size of the groups is indirectly controlled by the **ICET_MAGIC_K** environment variable or CMake variable.

ICET_SINGLE_IMAGE_STRATEGY_TREE At each phase, each process partners with another, and one of the processes sends its entire image to the other. The algorithm recurses with the group of processes that received images until only one process has an image.

A string documenting the current strategy can be retrieved with the **icetGetSingleImageStrategyName** function. The following sections describe the single image strategies in more detail.

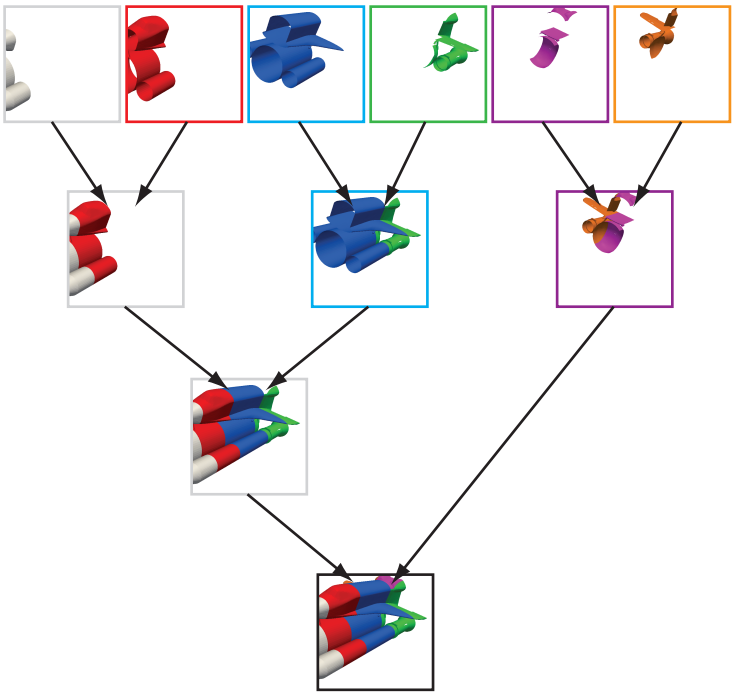

Figure 5.2. Tree composite network. Arrows represent the passing of data from one stage to the next. Processes receiving multiple images composite them together.

TREE COMPOSITING

The **tree composite** algorithm (sometimes also called binary tree composite due to its pairwise grouping) is a simple algorithm that iteratively combines full images together until they are all merged into a single image. The tree composite sub-strategy is selected by calling **`icetSingleImageStrategy`** with **`ICET_SINGLE_IMAGE_STRATEGY_TREE`**. The basic network for tree composite is shown in Figure 5.2.

The tree compositing algorithm is organized in stages. At each stage the processes pair up. One of the processes sends its data to its pair and then drops out of the computation. The receiving process combines the two images (using the compositing operation described in Chapter 4) and continues to the next stage. Processing continues until there is only one image (and one process) remaining.

As just defined, the tree composite algorithm only handles process counts that are a power of two (that is, the number of processes is equal to 2^i for some integer i). IceT handles non-powers of two gracefully. At any stage where the number of processes is not even and one of the processes cannot be paired, that leftover process does nothing for that stage but then continues to participate in the next stage. An example of this can be seen in the second stage of Figure 5.2.

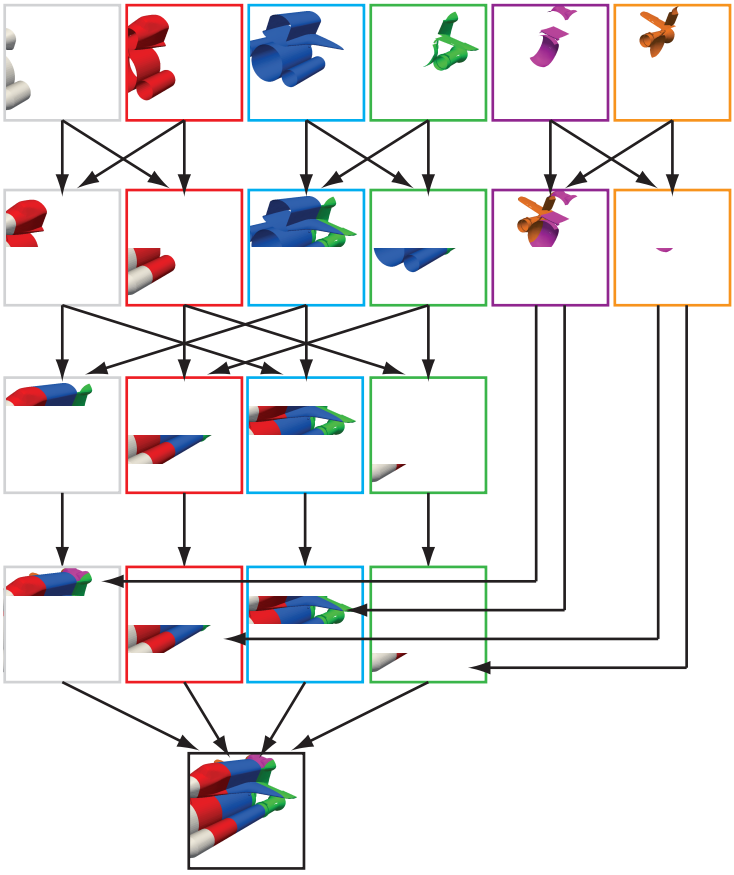

Figure 5.3. Binary-swap composite network. Arrows represent the passing of data from one stage to the next. Processes receiving multiple images composite or stitch them together. At most stages each process divides its image data and distributes it. The distribution of image data can be inferred from the target images.

The advantages of tree composite are its regular and efficient data transfers. The limiting factor of tree compositing is that at each stage of the algorithm half of the processes drop out of the computation. Thus, for more than a few processes tree compositing provides poor process utilization.

BINARY-SWAP COMPOSITING

The second single image compositing algorithm provided by IceT is the **binary-swap** algorithm. The binary-swap composite sub-strategy is selected by calling **`icetSingleImageStrategy`** with **`ICET_SINGLE_IMAGE_STRATEGY_BINARY_SWAP`**. The basic network for binary-swap composite is shown in Figure 5.3

Like tree compositing, binary swap is organized in stages, and at each stage the processes pair up. However, rather than have one process send all the data to the other, the image space is divided in two and the processes swap image data so that each process has all the data for part of the image. At the next stage, the processes pair up again, but with different partners that have the same partition of the image. Processing continues until each of the N processes have an image $1/N$ the size of the original image. At this point, all the processes send their sub-image to the display processes where the images are stitched together.

As just defined, the binary-swap composite algorithm only handles processes that are a power of two (that is, the number of processes is equal to 2^i for some integer i). Some binary-swap implementations handle non-powers of two by reducing the problem to the next largest power of two and dropping the leftover processes, but IceT handles non-powers of two more gracefully than that. Instead, IceT first finds the largest group of processes that is a power of two, makes a partition out of them, then finds the next largest group of processes that remain that is a power of two, makes a partition out of them, and so on. Each partition runs binary-swap independently up to the point where each process has its own piece of data. At this point, the smaller partitions send their image data to processes of the larger partitions, dividing up images where necessary. The largest partition then finishes the compositing in the normal way by collecting all of the pieces.

An example of compositing with a non-power of two is given in Figure 5.3. The six processes are partitioned first into a group of 4 and then into a group of 2. After swapping, the processes in the smaller group send images to the larger group. In this case, the purple process sends image data to the gray and blue processes, and the orange process sends to the red and green processes.

Like tree composite, binary swap exhibits regular and efficient data transfers. In addition, binary swap involves the use of all the processes throughout most of the compositing. Consequently, binary swap exhibits very good process utilization and scaling with respect to the number of processes on which it is run.

The most inefficient part of binary swap is the collection of image fragments at the end, which is an extra step that tree composite does not need to take. Most of the time the better parallel efficiency of binary swap over tree composite more than compensates for the extra collection step.

RADIX-K COMPOSITING

The third single image compositing algorithm provided by IceT is the **radix-k** algorithm. The radix-k composite sub-strategy is selected by calling **`icetSingleImageStrategy`** with **`ICET_SINGLE_IMAGE_STRATEGY_RADIXK`**. An example communication network for radix-k is shown in Figure 5.4.

Radix-k compositing is similar to binary swap. Both algorithms are organized in stages. However, where in binary swap processes pair up in groups of 2, radix-k groups processes in arbitrary (but consistent) groups of size k. Within each group, the image is split into k pieces, each piece is assigned to a process in the group, and all image pieces are sent to the assigned process. At the end of the stage, all processes with the same image piece are collected and recursed into. The

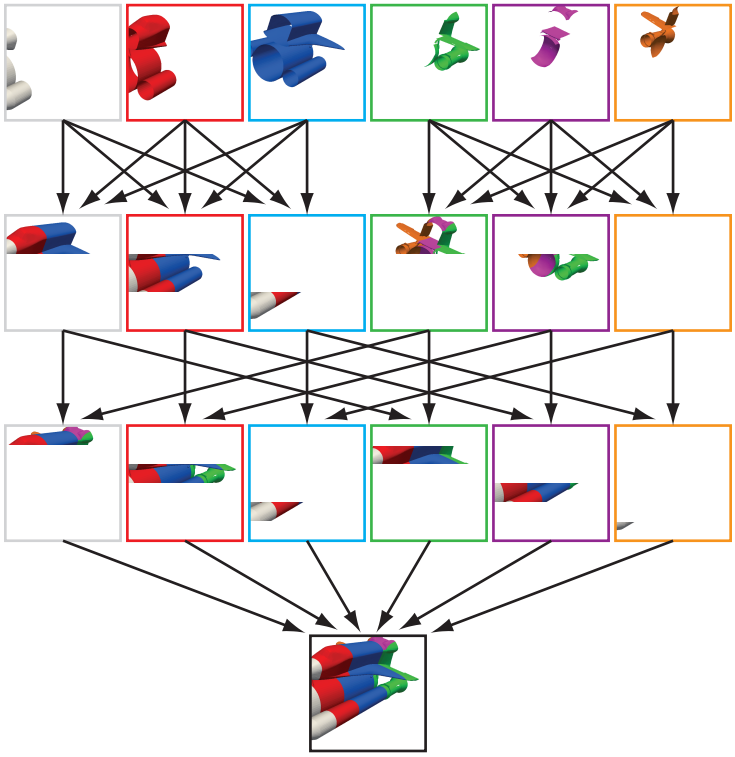

Figure 5.4. Radix-k composite network using a round of $k = 3$ and then a round of $k = 2$. Arrows represent the passing of data from one stage to the next. Processes receiving multiple images composite or stitch them together. The distribution of image data can be inferred from the target images.

binary-swap algorithm is a special case of radix-k with $k = 2$ for every stage.

Using radix-k with $k > 2$ offers two advantages. First, some high speed interconnects work more efficiently when there is more than one message being transferred over the network a time. Second, when a process is receiving more than one image piece at a time it has the opportunity to overlap pixel blending with the data transfer. As soon as the first image comes in it can be blended with the local image while the subsequent images are still in transit. However, the total number of messages created grows quadratically with k, so too large a value will make the communication less efficient.

The k value to use for each round is automatically determined by the number of processes participating in compositing. The k values can be indirectly controlled by setting the **ICET_MAGIC_K** environment variable. When set, IceT will pick k values as close as the **ICET_MAGIC_K** value as possible. If the **ICET_MAGIC_K** value is not set, then a hard-coded target k value is used, which can be set at compile time with the **ICET_MAGIC_K** CMake variable.

For more details on the radix-k algorithm, see the following paper.

Tom Peterka, David Goodell, Robert Ross, Han-Wei Shen, and Rajeev Thakur. "A Configurable Algorithm for Parallel Image-Compositing Applications," In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*, November 2009, DOI=10.1145/1654059.1654064.

AUTOMATIC ALGORITHM SELECTION

IceT also supports the automatic selection of the single image sub-strategy. This automatic selection is enabled by calling **icetSingleImageStrategy** with **ICET_SINGLE_IMAGE_STRATEGY_AUTOMATIC**. (It is also the default for the single image strategy.)

The automatic selection attempts to guess at the best strategy. The intension is that IceT can internally pick the best strategy depending on how the compositing is being used. For example, through some empirical studies, we found that the binary tree algorithm was more efficient than binary swap on less then 8 processes and less efficient on more than 8 processes. Consequently, IceT automatically switches between the two algorithms based on the amount of processes involved in the compositing.

ORDERED COMPOSITING

In some applications, the order in which images are composited together makes a difference (see the Volume Rendering section in Chapter 4). The details on how ordered compositing is achieved is not given here, but the basic idea for both compositing algorithms is that they first swizzle the processes so that their order matches the order in which the images need to be composited together. When compositing images together, they make sure to maintain over/under constancy based on the swizzled ranks from the originating processes. The networks are also managed such that no two images are composited that are not directly "next" to each other (that is, there is no image that needs to be inserted between them).

Reduce Strategy

An effective strategy implemented in IceT is the **reduce to single tile strategy** (or simply the reduce strategy). In this strategy, the multi-tile composite problem is efficiently reduced to a set of single image compositing problems, which are well studied and discussed in the previous section. The reduce strategy is selected by calling **icetStrategy** with the **ICET_STRATEGY_REDUCE** argument.

The reduce strategy is performed in two phases. In the first phase, processes are partitioned into groups, each of which is responsible for compositing the image of one of the tiles. The number of processes assigned to each tile is proportional to the number of non-empty images rendered for the

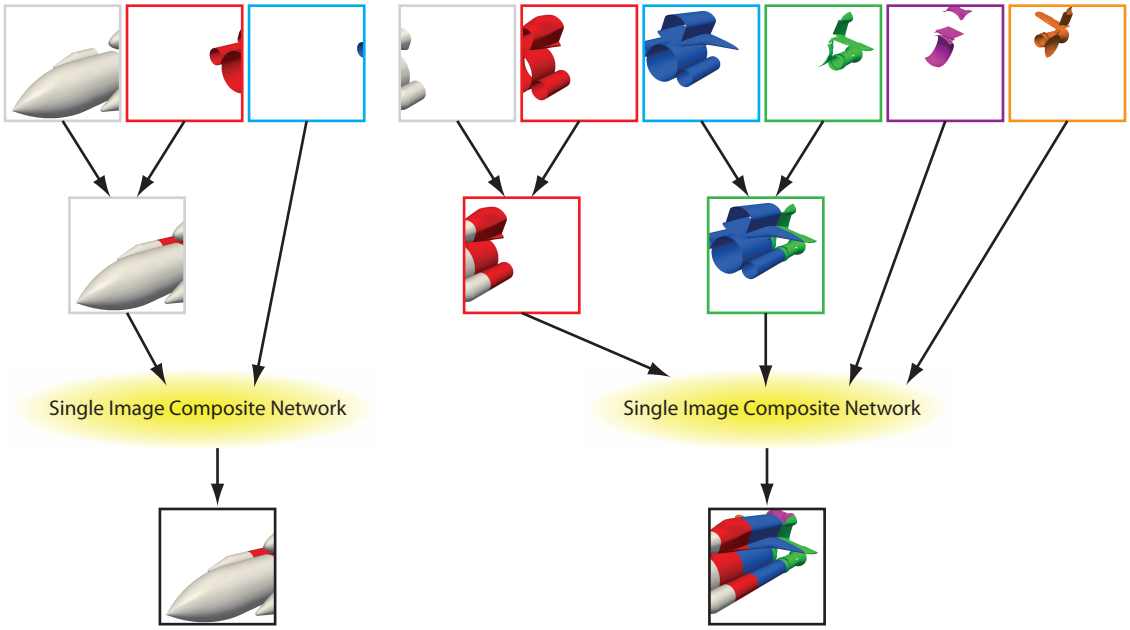

Figure 5.5. Composite network for reduce strategy. Arrows represent the passing of data from one stage to the next. Processes receiving multiple images composite them together. The single image composite network is described in a preceding section.

corresponding tile. In the example shown in Figure 5.5 there are a total of 9 non-empty images. The left tile has 3 of the 9, that is $\frac{1}{3}$, of the images and thus is assigned $\frac{1}{3} \times 6 = 2$ processes. Likewise, the right image is assigned $\frac{2}{3} \times 6 = 4$ processes.

When assigning processes to tiles, display processes and processes rendering images to the tile are given preference. In the example of Figure 5.5, the gray and blue processes are assigned to the left tile. The remainder are assigned to the right tile. Any image generated by a process that does not belong to the destination tile is transferred to a process assigned to the tile. In the example, the three processes that render two images, gray, red, and blue, each pass one of their images to a process in the opposing process group. All of these transfers have unique senders and receivers and thus can happen simultaneously.

In the second phase of the reduce strategy, each group of processes independently composites its images together using one of the single image compositing algorithms described in the preceding section.

The reduce strategy supports ordered compositing. It does this by ensuring in the first phase that processes receive only images that are "near" the image they hold, that is, there is no other image in between the two images in the visibility ordering. The single image compositing algorithms of the second phase each support their own ordered compositing.

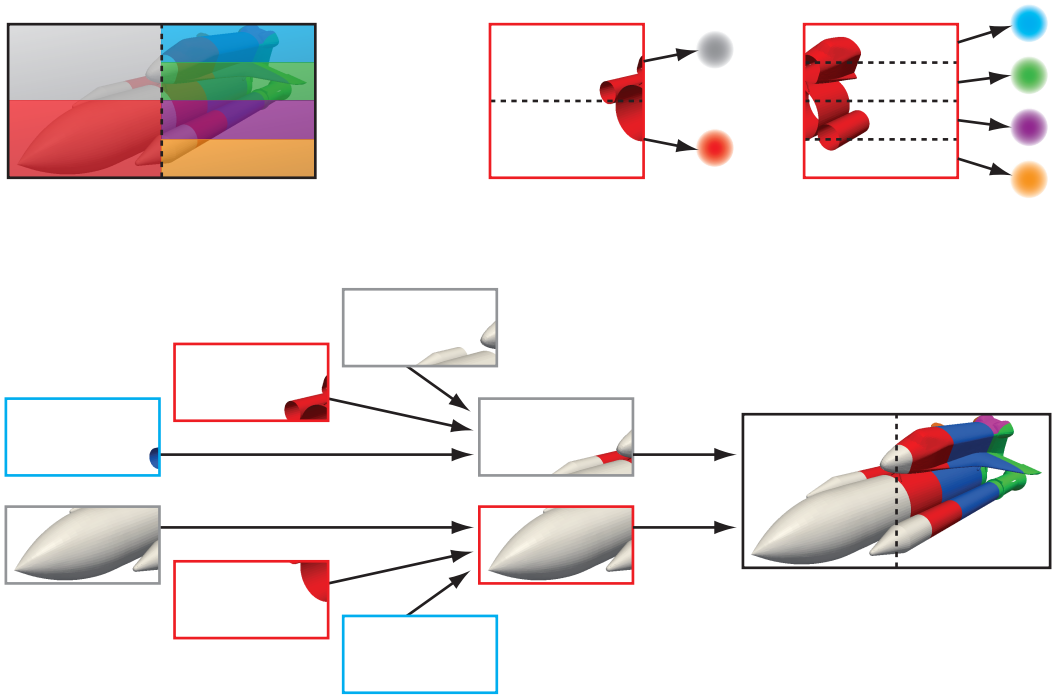

Figure 5.6. Compositing for split strategy. First tiles are split and assigned to processes (upper left). Then each process simultaneously sends its images to the responsible process (upper right) and receives all sub-images for its piece (bottom). The composited pieces are then collected and stitched together.

Split Strategy

The **tile split and delegate strategy** (or simply the split strategy) is a simple algorithm that splits up tiles, assigns each piece to a process, and then sends image fragments directly to the processes for compositing. The split strategy makes efficient use of processing resources, but exhibits haphazard and copious message passing which can cause issues on some high speed interconnects. The split strategy is selected by calling `icetStrategy` with the `ICET_STRATEGY_SPLIT` argument.

The split strategy first assigns processes to tiles similar to how they are assigned in the reduce strategy described previously. That is, the number of processes per tile is proportional to the number of non-empty images generated for it. Each tile is then split up evenly amongst all processes assigned to it. In the example in Figure 5.6, the upper left image shows that the left image is split between 2 processes and the right image is split amongst 4 processes.

On being assigned a section of tile, each process prepares to receive data from all the sending processes using asynchronous receives. Each process then renders its images, splits them up, and

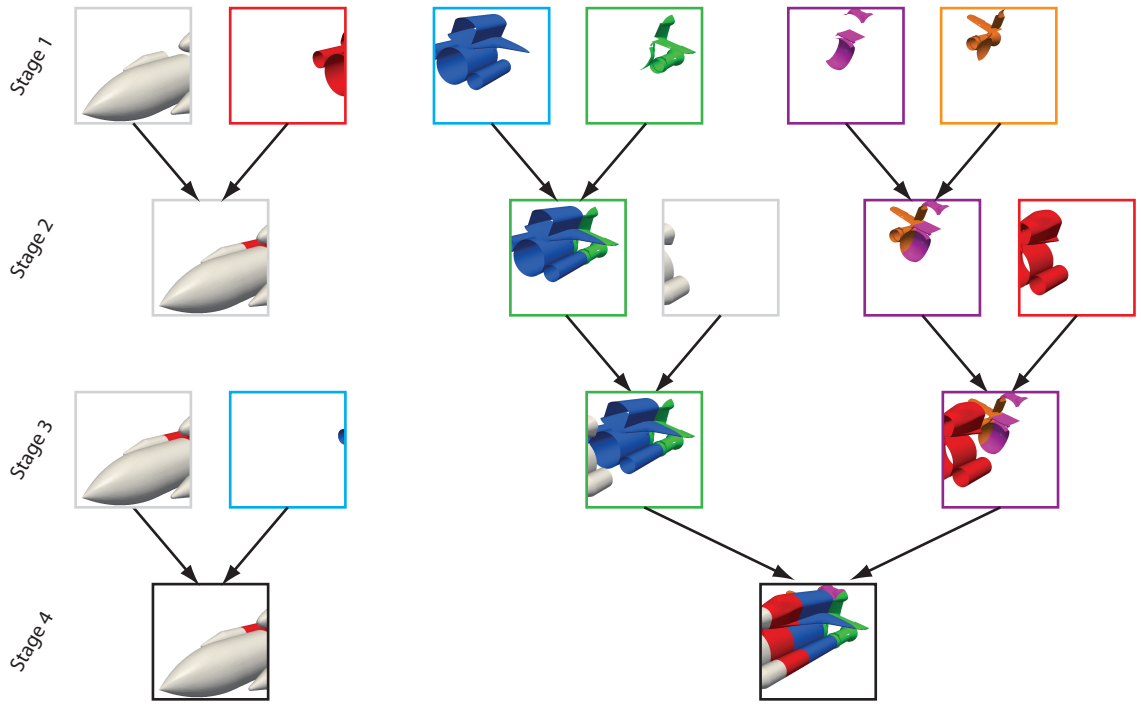

Figure 5.7. Composite network for virtual trees strategy. Arrows represent the passing of data from one state to the next. Processes receiving multiple images composite them together.

sends the sub-images to the corresponding process. When a process is ready and as it receives data, the incoming images are composited together. Once all of the incoming images are composited, the complete sub-image is sent to the display process to be stitched together.

The split strategy does not support ordered compositing. Using the split strategy in color blending mode will fail.

Virtual Trees Strategy

The **virtual trees strategy** is based on the binary tree compositing algorithm, but performs multiple composites simultaneously to regain some of the load balance lost in the original algorithm. The virtual trees strategy has nice regular communications, but still suffers from some load imbalance, particularly when using fewer tiles and in later stages of the algorithm. The virtual trees strategy is selected by calling **`icetStrategy`** with the **`ICET_STRATEGY_VTREE`** argument.

The virtual trees strategy works by creating a "virtual" tree for each tile. Contained in each tree are processes that have rendered an image for that display tile. The algorithm proceeds much like the binary tree composition algorithm except that the processes float amongst the trees, helping

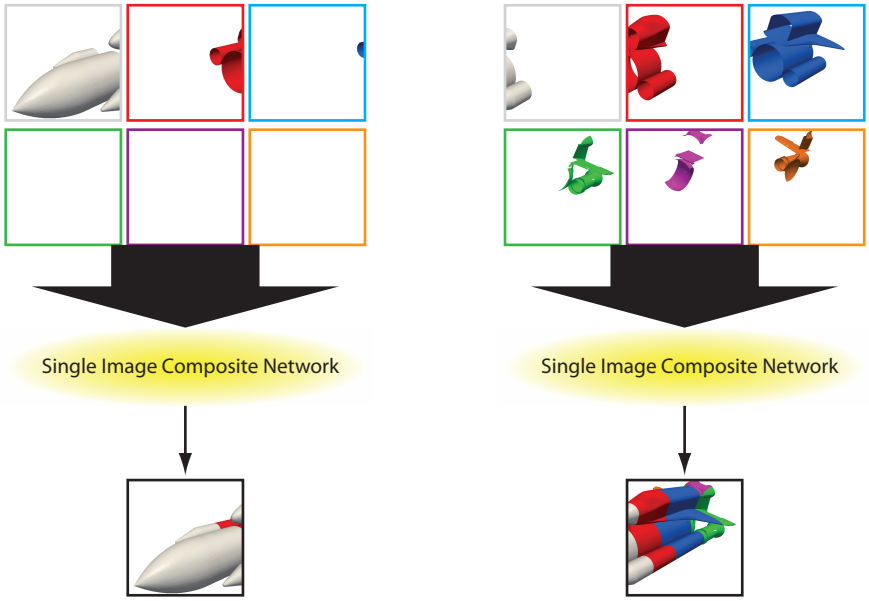

Figure 5.8. Composite network for sequential compositing. One at a time, each tile is composited using a parallel single image composite network described in a previous section.

with the compositing as they become available. Figure 5.7 shows an example of the virtual trees compositing. In particular, notice that the gray process takes part in the left tree in stage 1, then floats to take part in the right tree in stage 2, and then returns to take part in the left tree in stage 3.

When necessary, the process must keep track of multiple images belonging to different virtual trees. Two conserve memory, images are not rendered until they are needed. Also, a process can only hold two images at a time: one that it is sending and one that it is receiving. If a process is holding an image for one tile, it cannot receive an image for another tile until it sends away the image it is holding.

The virtual trees strategy does not support ordered compositing. Using the virtual trees strategy in color blending mode will fail.

Sequential Strategy

The **sequential strategy** sequentially addresses the tiles, but performs parallel compositing for each tile. The sequential strategy is selected by calling **`icetStrategy`** with the **`ICET_STRATEGY_SEQUENTIAL`** argument.

The sequential strategy iterates over all of the tiles. For each tile, it composites all the images

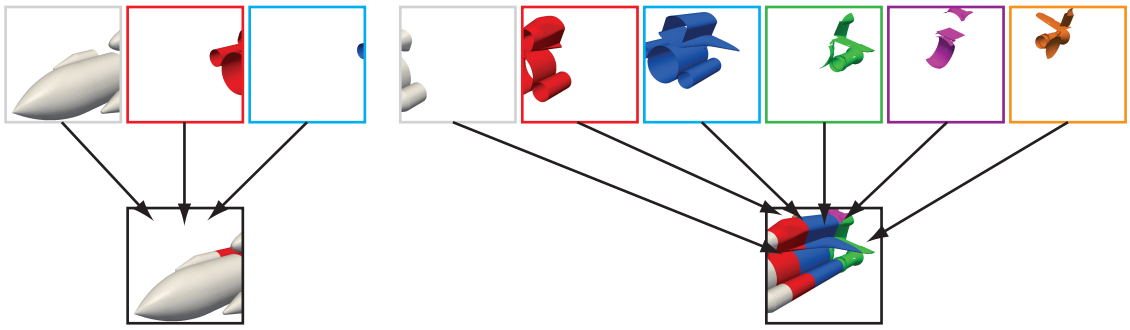

Figure 5.9. Composite network for direct send compositing. Arrows represent the passing of data from one process to another. Receiving process composite all incoming images together.

for that tile using one of the single image compositing algorithms described in that preceding section. As demonstrated in the example in Figure 5.8, images from all processes are composited for each tile regardless of whether some of them may be empty.

Since the single image compositing algorithms support ordered compositing, the sequential strategy also supports ordered compositing.

The sequential strategy is most useful in the special (but common) case of rendering to a single tile. In this case the sequential strategy can skip much of the global collective communication necessary for other strategies that must manage the sparse collection of tiles.

Direct Send Strategy

The **direct send strategy** is the simplest of all the strategies. Each process simply renders its images and sends them directly to the display process where the images get composited, as shown in Figure 5.9. The direct send strategy is selected by calling **`icetStrategy`** with the **`ICET_STRATEGY_DIRECT`** argument.

The direct send strategy is usually a poor performer. It was designed as a low watermark to compare to other compositing strategies. The direct send strategy does, however, support ordered compositing.

Chapter 6

Implementing New Strategies

The IceT API was written while its strategies were being developed. As such, the design yields for the relatively simplistic addition of both new multi-tile strategies and new single-image strategies. This chapter will provide the basic overview of how to add a new strategy for those interested in adding new compositing algorithms to IceT. It is probably easiest to start by modifying your IceT source to insert your own strategy in the `src/strategies` directory of the IceT source distribution.

A strategy in IceT is created by simply defining a function that performs the operation. A multi-tile strategy (one selected with **`icetStrategy`**) should take no arguments and return an **`IceTImage`**. Thus, a new multi-tile strategy function would look something like this. (The following sections will provide details on performing the individual tasks of the implementation.)

```
IceTImage icetCustomMultiTileCompose(void)
{
    /* Render images. */
    /* Transfer data. */
    /* Composite pixels. */
    /* Store results in image. */
    return image;
}
```

To expose the strategy from the IceT interface, add an identifier to `IceT.h` starting with `ICET_STRATEGY_` to the list of existing strategy identifiers. Then modify the functions in `src/strategies/select.c` to expose this new identifier to the rest of the IceT library. In particular, add your new identifier to the switch statements in the following functions.

`icetStrategyValid` Simply add your identifier to the list so that IceT can verify that your strategy is defined.

`icetStrategyNameFromEnum` Add a short human-readable name for your strategy. This is the string returned from **`icetGetStrategyName`**.

`icetStrategySupportsOrder` Return **`ICET_TRUE`** if your strategy can properly composite based on the ordering given in **`ICET_COMPOSITE_ORDER`**. Return **`ICET_FALSE`** otherwise. This value gets stored in the **`ICET_STRATEGY_SUPPORTS_ORDERING`**.

icetInvokeStrategy Call the function that invokes your strategy's image compositing (`icetCustomMultiTileCompose` in the example above).

The process for creating a single-image strategy (one selected with **icetSingleImageStrategy**) is similar. The first step is define a function that performs the compositing. However, the single-image composite function takes arguments that define the image to composite and the group of processes contributing. A new single-image strategy function would look something like this.

```
void icetCustomSingleImageCompose(const IceTInt *compose_group,
                                  IceTInt group_size,
                                  IceTInt image_dest,
                                  IceTSparseImage input_image,
                                  IceTSparseImage *result_image,
                                  IceTSizeType *piece_offset)
{
    /* Transfer data. */
    /* Composite pixels. */
    /* Point result_image to image object with results. */
    /* Store offset of local piece in piece_offset. */
}
```

The first argument, *compose_group*, is an array of process ranks. The pixels are to be composited in the order specified in this array. The second argument, *group_size*, specifies how many processes are contributing to the image and also specifies the length of *compose_group*.

The third argument, *image_dest*, specifies the process in which the final composed image should be placed. It is an index into *compose_group*, not the actual rank of the process. *image_dest* is really just a hint and can be ignored. The single image composite does not need to collect the composited pixels to a single process. It can (and usually does) return with pieces of the composited image distributed amongst nodes in the group. Any distribution is supported so long as the pieces are continuous, non-overlapping, and collectively define all the pixels in the image.

The fourth argument, *input_image*, contains the input image to be composited with the corresponding images in the other processes. The resulting image is returned via the final two arguments. *result_image* gets the sparse image object containing the composited image piece of the local process. It can have zero pixels if the local process has no image data. *piece_offset* gets the offset, in pixels, of the local image piece in the entire image.

To expose the single-image strategy from the IceT interface, add an identifier to `IceT.h` starting with `ICET_SINGLE_IMAGE_STRATEGY_` to the list of existing single-image strategy identifiers. Then modify the functions in `src/strategies/select.c` to expose this new identifier to the rest of the IceT library. In particular, add your new identifier to the switch statements in the following functions.

icetSingleImageStrategyValid Simply add your identifier to the list so that IceT can verify that your strategy is defined.

icetSingleImageStrategyNameFromEnum Add a short human-readable name for your strategy. This is the string returned from **icetGetSingleImageStrategyName**.

icetInvokeSingleImageStrategy Call the function that invokes your strategy's image compositing (`icetCustomSingleImageCompose` in the example above).

Internal State Variables for Compositing

The strategy compose functions are expected to get many of its parameters and other relevant information from the IceT state. Many of the relevant state variables are described in the documentation for the **icetGet** functions (as well as elsewhere throughout this document). There are also several "hidden" state variables for internal use. The ones specifically useful for within a composite function are listed here (along with the variable type, number of entries, and a description). Note that these state variables generally should be read from, not written to.

ICET_ALL_CONTAINED_TILES_MASKS (boolean, **ICET_NUM_TILES** $\times$ **ICET_NUM_PROCESSES**) Contains an appended list of **ICET_CONTAINED_TILES_MASK** variables for all processes. Given process p and tile t, the entry at $(\textbf{ICET\_NUM\_TILES} \times p) + t$ contains the flag describing whether process p renders a non-blank image for tile t. This variable is the same on all processes. This state variable is *not* set when using the sequential strategy.

ICET_CONTAINED_TILES_LIST (integer, **ICET_NUM_CONTAINED_TILES**) All the tiles into which the local geometry projects. In other words, this is the list of tiles which will not be empty after local rendering. The local processor should generate images for these tiles and participate in the composition of them.

ICET_CONTAINED_TILES_MASK (boolean, **ICET_NUM_TILES**) This is a list of boolean flags, one per tile. The flag is true if the local geometry projects onto the tile (that is, the local render will not be empty for that tile) and false otherwise. This gives the same information as **ICET_CONTAINED_TILES_LIST**, but in a different way that can be more convenient in some circumstances.

ICET_CONTAINED_VIEWPORT (integer, 4) Describes the region of the viewport that the geometry being rendered locally projects onto. The bounds of the data (given by **icetBoundingBox** or **icetBoundingVertices**) projected onto the tiled display determines the region of the tiled display the data covers. The values in the four-tuple correspond to x, y, width, and height, respectively, of the projection in global pixel coordinates. This variable in conjunction with the **ICET_NEAR_DEPTH** and **ICET_FAR_DEPTH** give the full 3D projection of the local data in window space.

ICET_FAR_DEPTH (double, 1) The maximum depth value of the local geometry after projection. See **ICET_CONTAINED_VIEWPORT** for more details.

ICET_IS_DRAWING_FRAME (boolean, 1) Set to true while in a call to **icetDrawFrame** or **icetGLDrawFrame** and set to false otherwise. This should always be set to true while the compose function is being executed.

ICET_MODELVIEW_MATRIX (double, 16) The current modelview matrix as passed to **icetDrawFrame** or read from OpenGL at the invocation of **icetGLDrawFrame**.

ICET_NEAR_DEPTH (double, 1) The minimum depth value of the local geometry after projection. See **ICET_CONTAINED_VIEWPORT** for more details.

ICET_NEED_BACKGROUND_CORRECTION (boolean, 1) If the resulting composited image needs to have its background corrected. That is, the final image should be blended with the color specified in **ICET_TRUE_BACKGROUND_COLOR** or **ICET_TRUE_BACKGROUND_COLOR**.

ICET_NUM_CONTAINED_TILES (integer, 1) The number of tiles into which the local geometry projects. This is the length of the **ICET_CONTAINED_TILES_LIST** variable.

ICET_PROJECTION_MATRIX (double, 16) The current projection matrix as passed to **icetDrawFrame** or read from OpenGL at the invocation of **icetGLDrawFrame**.

ICET_TILE_CONTRIB_COUNTS (integer, **ICET_NUM_TILES**) For each tile, provides the number of processes that will produce a non-empty image for that tile. This state variable is *not* set when using the sequential strategy.

ICET_TOTAL_IMAGE_COUNT (integer, 1) The total number of non-empty images produced by all processes for all tiles. This variable is the sum of all entries in **ICET_TILE_CONTRIB_COUNTS**. This state variable is *not* set when using the sequential strategy.

ICET_TRUE_BACKGROUND_COLOR (float, 4) An RGBA color identifying the "true" or final background color. If **ICET_NEED_BACKGROUND_CORRECTION** is true, then this color should be blended as the background to all pixels in the final image.

ICET_TRUE_BACKGROUND_COLOR_WORD (integer, 1) Same as **ICET_TRUE_BACKGROUND_COLOR** but stored as 8-bit values packed in an integer.

In addition to several internal state variables, IceT also has several internal functions for accessing them. The most important set for implementing a strategy is the **icetUnsafeStateGet** suite of functions, which are defined in the `IceTDevState.h` header file.

```
const IceTDouble *   icetUnsafeStateGetDouble(   IceTEnum  pname  );
const IceTFloat *    icetUnsafeStateGetFloat(    IceTEnum  pname  );
const IceTInt *      icetUnsafeStateGetInteger(  IceTEnum  pname  );
const IceTBoolean *  icetUnsafeStateGetBoolean(  IceTEnum  pname  );
const IceTVoid **    icetUnsafeStateGetPointer(  IceTEnum  pname  );
```

The implementation for the **icetGet** functions is to copy the data into a memory buffer you provide, performing type conversion as necessary. The **icetUnsafeStateGet** functions simply return the internal pointer where the data is stored. This can be faster and more convenient (since you do not have to allocate your own memory), but is unsafe in two ways. First, if the state variable is changed, the pointer you receive can become invalid. Second, no type conversion is performed. You have to make sure that you request a pointer of the correct type (or you will get an error). Since the state setting functions are hidden from the end user API, it is possible to manage these erroneous conditions. These functions return `const` pointers to discourage you from changing state values by directly manipulating the data in the pointers.

Memory Management

Compositing algorithms by their nature require buffers of memory of non-trivial size to hold images, among other data, that are not needed in between calls to the compositing. One approach is to simply use the standard C **malloc** and **free** functions. However, some implementations of **malloc**/**free** are not always efficient, and even the best implementations can have a tendency to fragment memory over time as large buffers are allocated and released.

During typical IceT operation, a strategy (whether it be a multi-tile strategy or a single-image strategy) is invoked multiple times. Each invocation will require multiple buffers to manipulate images and other data. One way to do this is to allocate these buffers as needed and free them by the end of the invocation. However, this can lead to the inefficiencies and memory fragmentation previously mentioned. It is also problematic when returning an image buffer as the responsibility for deallocating the buffer becomes undefined.

A better approach is to allocate the buffers as needed and then keep the buffers around for the next invocation of the strategy. This approach requires a certain amount of overhead to check when buffers need to be allocated or resized and when they can be freed. IceT uses its own state mechanism to assist in managing memory buffers. You do this by creating a **state buffer**, a buffer attached to a state variable. This is done with the **icetGetStateBuffer** function, which is defined in `IceTDevState.h`.

```
IceTVoid *icetGetStateBuffer(  IceTEnum      pname,
                               IceTSizeType  num_bytes  );
```

The **icetGetStateBuffer** takes a state variable and a buffer size in bytes. It then checks to see if a buffer of the appropriate size has already been allocated to that state variable. If so, it is returned. If not, a new buffer is allocated and returned. There are also similar functions called

icetGetStateBufferImage and **icetGetStateBufferSparseImage**, described in the following section, that allocate image buffers.

Because each buffer is assigned to a state variable, it is important to assign the buffer to a state variable that is both valid and unused by other IceT components. To this end, there are several state variables reserved for multi-tile strategies or single-image strategies. The state variables for multi-tile strategies are named **ICET_STRATEGY_BUFFER_***i* numbered from 0 to 15. That is, **ICET_STRATEGY_BUFFER_0**, **ICET_STRATEGY_BUFFER_1**, and so on up to **ICET_STRATEGY_BUFFER_15**.

By convention, your multi-tile strategy implementation should start by creating `#define` enumerations that alias these variables to logical names for the buffers. This will help prevent confusion or accidental sharing of buffers. Also by convention, try to make the largest buffers "first" (that is, **ICET_STRATEGY_BUFFER_0** has the largest buffer, **ICET_STRATEGY_BUFFER_1** has the next largest, and so on) so that if the strategy is changed, the large buffers will most likely be shared.

As an example, consider the following code taken from the virtual trees strategy implementation that aliases the buffer state variables it uses.

```
#define VTREE_IMAGE_BUFFER              ICET_STRATEGY_BUFFER_0
#define VTREE_IN_SPARSE_IMAGE_BUFFER    ICET_STRATEGY_BUFFER_1
#define VTREE_OUT_SPARSE_IMAGE_BUFFER   ICET_STRATEGY_BUFFER_2
#define VTREE_INFO_BUFFER               ICET_STRATEGY_BUFFER_3
#define VTREE_ALL_CONTAINED_TMASKS_BUFFER ICET_STRATEGY_BUFFER_4
```

And here is an example of these buffers being allocated.

```
sparseImageSize = icetSparseImageBufferSize(max_width, max_height);

image                = icetGetStateBufferImage(VTREE_IMAGE_BUFFER,
                                               max_width, max_height);
inSparseImageBuffer  = icetGetStateBuffer(VTREE_IN_SPARSE_IMAGE_BUFFER,
                                          sparseImageSize);
outSparseImage       = icetGetStateBufferSparseImage(
                                              VTREE_OUT_SPARSE_IMAGE_BUFFER,
                                              max_width, max_height);
info                 = icetGetStateBuffer(VTREE_INFO_BUFFER,
                                         sizeof(struct node_info)*num_proc);
all_contained_tmasks = icetGetStateBuffer(VTREE_ALL_CONTAINED_TMASKS_BUFFER,
                                    sizeof(IceTBoolean)*num_proc*num_tiles);
```

Once allocated, these buffers can be used and never need to be freed. IceT will handle the memory management. However, do not expect any of these buffers to contain the same data or even

exist on the next invocation of the strategy. Each invocation of the strategy should call **icetGetStateBuffer**, **icetGetStateBufferImage**, and **icetGetStateBufferSparseImage** to ensure that it has a valid buffer.

There is a separate set of state variables reserved for buffers used in single-image strategies. These are named **ICET_SI_STRATEGY_BUFFER_***i* numbered from 0 to 15. That is, **ICET_SI_STRATEGY_BUFFER_0**, **ICET_SI_STRATEGY_BUFFER_1**, and so on up to **ICET_SI_STRATEGY_BUFFER_15**. It is important *not* to use the multi-tile strategy buffer variables in a single-image strategy because the multi-tile strategy will call the single-image strategy while it is still operating and the single-image strategy can invalidate the buffers of the multi-tile strategy.

Image Manipulation Functions

IceT defines two image types: **IceTImage** and **IceTSparseImage**. Both image types can hold color data or depth data or both. The **IceTImage** type stores pixels as raw data, simple 2D arrays holding color or pixel data in horizontal-major order. The **IceTSparseImage** stores images using active-pixel encoding, the run length encoding described in the Active-Pixel Encoding section of Chapter 4.

Both the **IceTImage** type and the **IceTSparseImage** type are opaque to compositing algorithms. IceT provides functions for creating and manipulating images. Some of these functions are defined in `IceT.h` and exposed to user code. These exposed functions are documented in the Image Objects section of Chapter 3 starting on page 43. Other functions are protected from the user level code and reserved for use by the compositing algorithms and other parts of IceT. These functions are defined in `IceTDevImage.h` and are documented here. When creating a compositing strategy, be sure to include both of these header files.

CREATING IMAGES

The easiest and safest way to create an image is to use the **icetGetStateBufferImage** function (or **icetGetStateBufferSparseImage** for sparse images).

```
IceTImage icetGetStateBufferImage(  IceTEnum      pname,
                                    IceTSizeType  width,
                                    IceTSizeType  height  );

IceTSparseImage icetGetStateBufferSparseImage(
                                       IceTEnum      pname,
                                       IceTSizeType  width,
                                       IceTSizeType  height  );
```

Each of these functions allocates a state buffer (described in the previous section on Memory Management) for an image of size *width* by *height* on the given state variable (*pname*),

and returns an initialized image object. The image object is allocated and initialized for the color and depth formats specified by the **ICET_COLOR_FORMAT** and **ICET_DEPTH_FORMAT** state variables. Here is some code taken from the virtual trees strategy implementation that demonstrates the use of these functions.

```
image               = icetGetStateBufferImage(VTREE_IMAGE_BUFFER,
                                              max_width, max_height);
outSparseImage      = icetGetStateBufferSparseImage(
                                            VTREE_OUT_SPARSE_IMAGE_BUFFER,
                                            max_width, max_height);
```

After an image is allocated, it is possible to resize the image, but only to dimensions that are less than or equal to those for which the image was created. This is done with the **icetImageSetDimensions** or **icetSparseImageSetDimensions** function.

```
void icetImageSetDimensions( IceTImage    image,
                             IceTSizeType width,
                             IceTSizeType height  );

void icetSparseImageSetDimensions( IceTSparseImage image,
                                   IceTSizeType    width,
                                   IceTSizeType    height  );
```

These functions do not resize the internal buffer of the image. Rather, they set the internal width and height parameters of the image and reuse the original (and potentially larger than necessary) buffer. This is why they cannot be used to size the image larger than the original buffer allocation. It is for this reason that it is typical for a multi-tile strategy to create images of size **ICET_TILE_MAX_WIDTH** and **ICET_TILE_MAX_HEIGHT**. A well designed compositing algorithm should never need more space than that. Likewise, it is typical for a single-image strategy to create images of the same size as the input image.

It is sometimes necessary to know the size of buffer required to store image data. This most often occurs when allocating buffers to receive images (as described in detail in the following section on Transferring Images starting on page 98). Getting the necessary buffer size is done with the **icetImageBufferSize** and **icetSparseImageBufferSize** functions.

```
IceTSizeType icetImageBufferSize( IceTSizeType width,
                                  IceTSizeType height  );

IceTSizeType icetSparseImageBufferSize( IceTSizeType width,
                                        IceTSizeType height  );
```

Each of these functions returns the *maximum* number of bytes required to store the image of the given dimensions and the formats specified by the **ICET_COLOR_FORMAT** and **ICET_DEPTH_FORMAT** state variables.

It is also possible, although discouraged, to convert a previously allocated buffer into an image object with one of the following functions.

```
IceTImage icetImageAssignBuffer( IceTVoid *    buffer,
                                 IceTSizeType  width,
                                 IceTSizeType  height  );

IceTSparseImage icetSparseImageAssignBuffer(
                                      IceTVoid *    buffer,
                                      IceTSizeType  width,
                                      IceTSizeType  height  );
```

In each case it is assumed that the buffer is at least as large as that indicated by the **icetImageBufferSize** or **icetSparseImageBufferSize** function.

IceT also defines a special **null image** that can be used as a place holder when no image data is available. Null images for both regular and sparse images are available. They are retreived with the following functions.

```
IceTImage icetImageNull( void );

IceTSparseImage icetSparseImageNull( void );
```

QUERYING IMAGES

IceT contains several functions that allow you to query basic information about an image object such as dimensions and data formats. Each function takes an information object and returns the appropriate size or identifier. (More detail for the functions that work on **IceTImage** objects is given in the Image Objects section of Chapter 3 starting on page 43.)

```
IceTSizeType  icetImageGetWidth       ( const IceTImage  image );
IceTSizeType  icetImageGetHeight      ( const IceTImage  image );
IceTSizeType  icetImageGetNumPixels   ( const IceTImage  image );

IceTEnum  icetImageGetColorFormat ( const IceTImage  image );
IceTEnum  icetImageGetDepthFormat ( const IceTImage  image );

IceTSizeType icetSparseImageGetWidth(
                                  const IceTSparseImage  image );
IceTSizeType icetSparseImageGetHeight(
                                  const IceTSparseImage  image );
IceTSizeType icetSparseImageGetNumPixels(
                                  const IceTSparseImage  image );
```

```
IceTEnum icetSparseImageGetColorFormat(
                                    const IceTSparseImage  image  );
IceTEnum icetSparseImageGetDepthFormat(
                                    const IceTSparseImage  image  );
```

IceT also provides several functions for retrieving data from **IceTImage** objects. These functions are described in the Image Objects section of Chapter 3 starting on page 43. There is no mechanism for directly accessing the data in an **IceTSparseImage**. Instead, data is indirectly manipulated via compression and compositing functions, which are described in the subsequent sections.

As implied in the previous section on creating images, an **IceTImage** or **IceTSparseImage** object has a pointer to a buffer containing the actual image data. It is sometimes helpful to determine if two images have the same buffer.

```
IceTBoolean icetImageEqual(  const IceTImage  image1,
                             const IceTImage  image1  );

IceTBoolean icetSparseImageEqual(  const IceTSparseImage  image1,
                                   const IceTSparseImage  image1  );
```

Both **icetImageEqual** and **icetSparseImageEqual** take two image objects and returns whether these two images point to the same buffer. If two images are equal, then changing the pixel data of one image also changes the data of the other. Two images may have the same data but still be considered different if they have separate buffers.

There are also special functions for testing whether an image is the null image.

```
IceTBoolean icetImageIsNull(  IceTImage  image  );

IceTBoolean icetSparseImageIsNull(  IceTSparseImage  image  );
```

SETTING PIXEL DATA

There are several mechanisms for setting, changing, or copying pixel data in **IceTImage** objects. Foremost are the **icetImageGetColor** and **icetImageGetDepth** functions that return the data buffer containing the color or depth values.

```
IceTUByte *  icetImageGetColorub  (  IceTImage  image  );
IceTUInt *   icetImageGetColorui  (  IceTImage  image  );
IceTFloat *  icetImageGetColorf   (  IceTImage  image  );

IceTFloat *  icetImageGetDepthf   (  IceTImage  image  );
```

The pointers returned from these functions are shared with the **IceTImage** object itself, so writing data into the buffer will change the image object as well.

There are no equivalent mechanisms for changing pixel data in **IceTSparseImage** objects. Instead, data is indirectly manipulated via compression, copy, and compositing functions, which are described in the subsequent sections.

It is fairly common to need to clear an image. This is common in a multi-tile strategy when returning an image for which no geometry is rendered. IceT provides convenience functions for setting all the data in an image to the background color.

```
void icetClearImage( IceTImage image );

void icetClearSparseImage( IceTSparseImage image );
```

COPYING FULL PIXEL DATA

It is common to need to copy pixel data from one image to another. IceT provides multiple helper functions to copy data amongst images. The first function is **icetImageCopyPixels**, which copies a contiguous section of pixels.

```
void icetImageCopyPixels( const IceTImage  in_image,
                          IceTSizeType     in_offset,
                          IceTImage        out_image,
                          IceTSizeType     out_offset,
                          IceTSizeType     num_pixels );
```

icetImageCopyPixels copies pixel data from *in_image* to *out_image*. *in_image* and *out_image* must have the same format. Both color and depth values are copied when available. The data is taken from the input starting at index *in_offset* and are placed in the output starting at index *out_offset*. *num_pixels* are copied in all. The following example code copies the entire contents from in_image to out_image (assuming they have the same sizes and formats).

```
icetImageCopyPixels(in_image, 0, out_image, 0, icetImageGetNumPixels(in_image));
```

This example copies the third row of data from the input image to the fifth row of data in the output image.

```
width = icetImageGetWidth(in_image);
icetImageCopyPixels(in_image, 3*width, out_image, 5*width, width);
```

This example copies the second half of pixels in in_image and places it in the first part of out_image. Notice that coping a contiguous region of pixels makes it easy to divide images in halves (or thirds, or whatevers), which is a common operation in image compositing, without having to worry about image dimensions.

```
num_pixels = icetImageGetNumPixels(in_image);
icetImageCopyPixels(in_image, num_pixels/2, out_image, 0, num_pixels/2);
```

A second convenience function for copying data amongst arrays is **icetImageCopyRegion**. This function works much like **icetImageCopyPixels**, except that you specify a rectangular 2D viewport window to copy rather than a 1D array region of pixels.

```
void icetImageCopyRegion(  const IceTImage  in_image,
                           const IceTInt *  in_viewport,
                           IceTImage        out_image,
                           const IceTInt *  out_viewport  );
```

in_viewport is an array containing 4 values that specify the rectangular region from which to copy. The first 2 values specify the x and y position of the lower left corner of the region. The second 2 values specify the width and height of the region. *out_viewport* is a similar array that specifies the destination region. The width and height of *in_viewport* and *out_viewport* must be the same. Here is a simple example of copying all the pixels from `in_image` to `out_image`.

```
IceTInt full_viewport[4];

full_viewport[0] = 0;
full_viewport[1] = 0;
full_viewport[2] = icetImageGetWidth(in_image);
full_viewport[3] = icetImageGetHeight(out_image);

icetImageCopyRegion(in_image, full_viewport, out_image, full_viewport);
```

This example copies a 50×50 region of pixels from the lower left corner of `in_image` to the upper right corner of `out_image`.

```
IceTInt in_viewport[4], out_viewport[3];

in_viewport[0] = 0;   in_viewport[1] = 0;
in_viewport[2] = 50;  in_viewport[3] = 50;

out_viewport[0] = icetImageGetWidth(in_image) - 50;
out_viewport[1] = icetImageGetHeight(in_image) - 50;
out_viewport[2] = 50;
out_viewport[3] = 50;

icetImageCopyRegion(in_image, in_viewport, out_image, out_viewport);
```

As mentioned previously, there is a **icetClearImage** function to clear the contents of an image to background. There is also another function called **icetImageClearAroundRegion** that sets the image to background everywhere but in a specified 2D viewport window.

```
void icetImageClearAroundRegion( IceTImage       image,
                                 const IceTInt * region  );
```

Expanding on the previous example, here is code that copies a region of pixels and then clears everything outside of this region in the destination.

```
icetImageCopyRegion(in_image, in_viewport, out_image, out_viewport);
icetImageClearAroundRegion(out_image, out_viewport);
```

COPY SPARSE IMAGE DATA

IceT also conatins several functions for efficiently copying pixels in sparse images. Because of the differences in implementation and use, the copy functions differ significantly between the full image and sparse image copy functions.

Basic Sparse Image Copy

```
void icetSparseImageCopyPixels( const IceTSparseImage  in_image,
                                IceTSizeType           in_offset,
                                IceTSizeType           num_pixels,
                                IceTSparseImage        out_image);
```

icetSparseImageCopyPixels copies a region of continuous pixels from *in_image* to *out_image*. The region starts a pixel offset *in_offset* and contains *num_pixels*. *out_image* is resized to contain only the copied pixels. The new size will have a width of *num_pixels* and a height of 1.

Sparse Image Split

Parallel compositing algorithms often involve splitting an image into a number of (approximately) equal sized pieces and distributing them amongst processes. This can be achieved by iteratively calling **icetSparseImageCopyPixels**. However, each call to **icetSparseImageCopyPixels** has to search through the pixels in *in_image* to the appropriate *in_offset*. It is a bit more efficient (and convenient) to iterate over the image once and copy to multiple different output images as you go. The **icetSparseImageSplit** function does just that.

```
void icetSparseImageSplit(
                    const IceTSparseImage  in_image,
                    IceTSizeType           in_image_offset,
                    IceTInt                num_partitions,
                    IceTInt                eventual_num_partitions,
                    IceTSparseImage *      out_images,
                    IceTSizeType *         offsets  );
```

icetSparseImageSplit takes *in_image*, partitions it into *num_partitions*, and stores the results in the array of pre-allocated images *out_images*. As an optimization, the image in the first index of *out_images* may be the same as *in_image*. In this case the first partition will be copied "in place." It is an error to have *in_image* in any other index of *out_images*. The offset of each image with respect to the original image is stored in the corresponding index of the array *offsets*.

The *in_image_offset* and *eventual_num_partitions* are for recursive splits, described in the following section. For a single invocation of **icetSparseImageSplit** for an image, *in_image_offset* and *eventual_num_partitions* should be set to 0 and the same value as *num_partitions*, respectively.

Before calling **icetSparseImageSplit**, it is important to allocate images with enough pixel space. To allocate these images, you first need to know how big each partition will be. **icetSparseImageSplitPartitionNumPixels** returns the maximum size of each partition in pixels. Do not assume that a partition size will be the total number of pixels divided by the number of partitions. This assumption is wrong when the number of pixels in the original image does not divide by the number of partitions.

```
void icetSparseImageSplitPartitionNumPixels(
                              IceTSizeType  input_num_pixels,
                              IceTInt       num_partitions,
                              IceTInt       eventual_num_partitions  );
```

Because the number of partitions may be large or unknown at compile time, it can be problematic to fill the array of output images to **icetSparseImageSplit** with images created with **icetGetStateBufferSparseImage** due to the limited number of available state variables. In this case, it is prudent to create a large enough buffer with **icetGetStateBuffer** and break it up into pieces to make sparse image objects with **icetSparseImageAssignBuffer**. The following code gives an example of using **icetSparseImageSplit**. This example uses copy-in-place for the first partition, but a trivial change makes a copy to this buffer.

```
#define NUM_SPLITS 8

/* original_image is image to be split. */

original_num_pixels = icetSparseImageGetNumPixels(original_pixels);
partition_num_pixels
```

```
    = icetSparseImageSplitPartitionNumPixels(original_num_pixels,
                                             NUM_SPLITS,
                                             NUM_SPLITS);
partition_buffer_size = icetSparseImageBufferSize(partition_num_pixels, 1);

split_image_buffer = icetGetStateBuffer(MYCOMPOSITE_SPLIT_IMAGE_BUFFER,
                                        (NUM_SPLITS-1)*partition_buffer_size);

out_images[0] = original_image;
for (i = 1; i < NUM_SPLITS; i++) {
    out_images[i] = icetSparseImageAssignBuffer(split_image_buffer,
                                                partition_num_pixels, 1);
    split_image_buffer += partition_buffer_size;
}

icetSparseImageSplit(original_image,
                     0,
                     NUM_SPLITS,
                     NUM_SPLITS,
                     out_images,
                     offsets);

for (i = 0; i < NUM_SPLITS; i++) {
    DoSomething(out_images[i], offsets[i]);
}
```

Recursive Sparse Image Split

Some image compositing algorithms, such as binary swap and radix-k, recursively split their images in subsequent rounds. It is also sometimes the case, such as when telescoping, that different processes will split images with different factors. For example, one process might split its image into eight pieces with three recursive calls of two partitions while another process creates the same partition with one split of two partitions and another split of four partitions while yet another makes one split of eight partitions.

Regardless of how the image is split, it is often necessary for the final partitions to match exactly with respect to offset and size. Unfortunately, if the size of the original image is not evenly divisible by the eventual number of partitions, different recursive partitions could lead to different image pieces.

To get around this problem, **icetSparseImageSplit** has the *in_image_offset* and *eventual_num_partitions* arguments. The *in_image_offset* declares that the *in_image* came from a previous call to **icetSparseImageSplit** with the given offset. The *eventual_num_partitions* argument declares the total number of partitions that will be made with this call to **icetSparseImageSplit** and all subsequent calls to **icetSparseImage-**

Split. Obviously, *num_partitions* must be a factor of *eventual_num_partitions*, or otherwise *eventual_num_partitions* could never be created. As long as the recursive calls to **icetSparseImageSplit** are consistent with the *in_image_offset* and *eventual_num_partitions* arguments, the partitions will match up exactly.

The following code example will result in the exact same image partitions as those in the previous example, but does it with two recursive calls.

```
#define FIRST_NUM_SPLITS 2
#define SECOND_NUM_SPLITS 4
#define TOTAL_NUM_SPLITS (FIRST_NUM_SPLITS * SECOND_NUM_SPLITS)

/* original_image is image to be split. */

/* Perform first level image split. */
original_num_pixels = icetSparseImageGetNumPixels(original_pixels);
partition_num_pixels
    = icetSparseImageSplitPartitionNumPixels(original_num_pixels,
                                             FIRST_NUM_SPLITS,
                                             TOTAL_NUM_SPLITS);
partition_buffer_size = icetSparseImageBufferSize(partition_num_pixels, 1);

split_image_buffer = icetGetStateBuffer(
                                    MYCOMPOSITE_FIRST_SPLIT_IMAGE_BUFFER,
                                    (FIRST_NUM_SPLITS-1)*partition_buffer_size);

intermediate_images[0] = original_image;
for (i = 1; i < FIRST_NUM_SPLITS; i++) {
    intermediate_images[i] = icetSparseImageAssignBuffer(split_image_buffer,
                                                         partition_num_pixels,
                                                         1);
    split_image_buffer += partition_buffer_size;
}

icetSparseImageSplit(original_image,
                     0,
                     FIRST_NUM_SPLITS,
                     TOTAL_NUM_SPLITS,
                     intermediate_images,
                     interpediate_offsets);

/* Perform second level image split. */
for (j = 0; j < FIRST_NUM_SPLITS; j++) {
    intermediate_num_pixels = icetSparseImageGetNumPixels(original_pixels);

    partition_num_pixels = icetSparseImageSplitPartitionNumPixels(
                                             original_num_pixels,
```

```
                                        SECOND_NUM_SPLITS,
                                        TOTAL_NUM_SPLITS/FIRST_NUM_SPLITS);
    partition_buffer_size = icetSparseImageBufferSize(partition_num_pixels, 1);

    split_image_buffer = icetGetStateBuffer(
                                MYCOMPOSITE_FIRST_SPLIT_IMAGE_BUFFER,
                                (SECOND_NUM_SPLITS-1)*partition_buffer_size);

    out_images[0] = original_image;
    for (i = 1; i < SECOND_NUM_SPLITS; i++) {
        out_images[i] = icetSparseImageAssignBuffer(split_image_buffer,
                                                    partition_num_pixels, 1);
        split_image_buffer += partition_buffer_size;
    }

    icetSparseImageSplit(intermediate_images[j],
                         intermediate_offsets[j],
                         FIRST_NUM_SPLITS,
                         TOTAL_NUM_SPLITS/FIRST_NUM_SPLITS,
                         out_images,
                         out_offsets);

    for (i = 0; i < SECOND_NUM_SPLITS; i++) {
        DoSomething(out_images[i], out_offsets[i]);
    }
}
```

This example is artificial in that the recursive splits are unlikely to be done on all partitions. Typical operation is to split an image and then send all images but one to other processes. Generally, images of the kept partition are also collected from other processes and blended. The recursive split then happens only on that one partition kept.

Interlacing Images

Strategies that use **icetSparseImageSplit** should consider honoring the **ICET_INTERLACE_IMAGES** option. Image interlacing is described in Chapter 4 starting on page 55. IceT provides a pair of functions, **icetSparseImageInterlace** and **icetGetInterlaceOffset**, to simplify this.

```
void icetSparseImageInterlace(
                    const IceTSparseImage  in_image,
                    IceTInt                eventual_num_partitions,
                    IceTEnum               scratch_state_buffer,
                    IceTSparseImage        out_image  );
```

icetSparseImageInterlace copies all the pixels from *in_image* to *out_image*, but shuffling the pixels around. The pixel shuffling is done in such a way that if you subsequently split the image with one or more calls to **icetSparseImageSplit** to create *eventual_num_partitions* (using the appropriate recursive calling described previously as necessary), then all the resulting image partitions will contain a continuous array of pixels. **icetSparseImageInterlace** requires a temporary buffer during its operation. It thus requires an identifier to a state variable for a buffer not being used.

Once an interlaced image is completely split, no further pixel shuffling is necessary. However, because the partitions have been shuffled, the offsets that are reported by **icetSparseImageInterlace** are incorrect. The correct offset is retrieved with **icetGetInterlaceOffset**

```
IceTSizeType icetGetInterlaceOffset(
                                IceTInt       partition_index,
                                IceTInt       eventual_num_partitions,
                                IceTSizeType  original_image_size );
```

icetGetInterlaceOffset gets information about the original image and a particular partition and returns the actual offset of that partition. *partition_index* is the index of the partition (the same as those used by a call to **icetSparseImageSplit** that was called non-recursively). *eventual_num_partitions* is the same as that in the call to **icetSparseImageInterlace** and the first call to **icetSparseImageSplit**. *original_image_size* is the number of pixels in the starting image before it was split.

The following code is boilerplate for implementing image interlacing in a single image strategy.

```
#define MY_COMPOSE_INTERLACE_IMAGE      ICET_SI_STRATEGY_BUFFER_0
#define MY_COMPOSE_DUMMY_ARRAY          ICET_SI_STRATEGY_BUFFER_1
/*...*/

void icetMySingleImageCompose(IceTInt *compose_group,
                              IceTInt group_size,
                              IceTInt image_dest,
                              IceTSparseImage input_image,
                              IceTSparseImage *result_image,
                              IceTSizeType *piece_offset)
{
    IceTSizeType original_image_size;
    IceTInt eventual_num_partitions;
    IceTBoolean use_interlace;
    IceTSparseImage working_image;
    IceTSparseImage final_image;
    IceTInt my_piece_index;
    IceTInt my_piece_offset;

    original_image_size = icetSparseImageGetNumPixels(input_image);
```

```
    eventual_num_partitions = /* Num total partitions to be created. */;

    use_interlace  = icetIsEnabled(ICET_INTERLACE_IMAGES);
    use_interlace &= (eventual_num_partitions > 2);
    if (use_interlace) {
        working_image = icetGetStateBufferSparseImage(
                                        MY_COMPOSE_INTERLACE_IMAGE,
                                        icetSparseImageGetWidth(input_image),
                                        icetSparseImageGetHeight(input_image));
        icetSparseImageInterlace(input_image,
                                 eventual_num_partitions,
                                 MY_COMPOSE_DUMMY_ARRAY,
                                 working_image);
    } else {
        working_image = input_image;
    }

    /* Do image compositing.  Set final_image to my piece to be returned.
       Set my_piece_index to the index of the partition in final_image.
       Set my_piece_offset to the appropriate offest returned from
       icetSparseImageSplit. */

    *result_image = final_image;
    if (use_interlace) {
        *piece_offset = icetGetInterlaceOffset(my_piece_index,
                                               eventual_num_partitions,
                                               original_image_size);
    } else {
        *piece_offset = my_piece_offset;
    }
}
```

COMPRESSING IMAGES

icetCompressImage converts a full **IceTImage** into to more compact **IceTSparseImage**.

```
void icetCompressImage(  const IceTImage       image,
                         IceTSparseImage       compressed_image);
```

Sometimes it is convenient to break up an image into pieces, and compress each piece. This is common when dividing up an image to be divvied amongst some amount of processes. This can be most easily achieved by using the **icetCompressSubImage**.

```
void icetCompressSubImage( const IceTImage   image,
                           IceTSizeType      offset,
                           IceTSizeType      pixels,
                           IceTSparseImage   compressed_image );
```

icetCompressSubImage compresses a region of contiguous pixels. The block of pixels starts *offset* pixels past the beginning of the image and is *pixels* long. *icetCompressImage* is almost equivalent to calling *icetCompressSubImage* with *offset* set to 0 and *pixels* set to the result of **icetImageGetNumPixels**. When compressing an image with **icetCompressSubImage**, the output **IceTSparseImage** has its width set to the *pixels* argument and its height set to 1.

A sparse image can be returned to its uncompressed form with **icetDecompressImage** or **icetDecompressSubImage**.

```
void icetDecompressImage(
                        const IceTSparseImage  compressed_image,
                        IceTImage              image  );

void icetDecompressSubImage(
                        const IceTSparseImage  compressed_image,
                        IceTSizeType           offset,
                        IceTImage              image  );
```

RENDERING IMAGES

A multi-tile compositing strategy is responsible for rendering images on demand as well as compositing. To render and retrieve the image for a particular tile in the display, use either **icetGetTileImage** or **icetGetCompressedTileImage**.

```
void icetGetTileImage( IceTInt     tile,
                       IceTImage   image  );

void icetGetCompressedTileImage( IceTInt           tile,
                                 IceTSparseImage   image  );
```

Both functions will invoke a rendering for that tile (performing the appropriate projection transformations) as necessary, read back the frame buffers and store the results in an image buffer you specify. The difference, of course, is that **icetGetTileImage** fills the buffer with raw data whereas **icetGetCompressedTileImage** will compress the image data with active-pixel encoding.

Although it is roughly equivalent to calling **icetGetTileImage** and then **icetCompressImage**, **icetGetCompressedTileImage** can be much more efficient.

IMAGE COMPOSITING

The IceT library contains multiple functions to locally composite two images together. These functions handle the complexities of dealing with different image formats and compositing operations.

```
void icetComposite(  IceTImage       destBuffer,
                     const IceTImage srcBuffer,
                     int             srcOnTop  );
```

icetComposite takes the images stored in *destBuffer* and *srcBuffer*, composites them together, and stores the result in *destBuffer*. The compositing operation is determined by the **ICET_COMPOSITE_MODE** state variable. (See the discussion on Compositing Operations in Chapter 4 for information on how the compositing operation is determined.) If the compositing operation is order dependent, then the Boolean argument *srcOnTop* determines whether *srcBuffer* or *destBuffer* is on top.

If one of your images is compressed (stored in an **IceTSparseImage**, it is faster to perform the compositing operation on the compressed image rather than decompressing first. In fact, it is faster to composite a compressed image than two full images because the active-pixel encoding allows the composite algorithm to skip over groups of background pixels. This gives you the double win of faster image transfer and faster compositing.

```
void icetCompressedComposite(
                            IceTImage             destBuffer,
                            const IceTSparseImage srcBuffer,
                            int                   srcOnTop  );
```

icetCompressedComposite behaves just like **icetComposite** except that *srcBuffer* is a compressed image rather than a full image. The images in *destBuffer* and *srcBuffer* are composited together, and the results are stored in *destBuffer*.

Many parallel compositing algorithms break images into pieces, distribute amongst processes, and composite the pieces. To facilitate the compositing image pieces, IceT provides **icetCompressedSubComposite**.

```
void icetCompressedSubComposite(
                            IceTImage             destBuffer,
                            IceTSizeType          offset,
                            const IceTSparseImage srcBuffer,
                            int                   srcOnTop);
```

The *destBuffer*, *srcBuffer* and *srcOnTop* arguments are the same as those in **icetCompressedComposite**. The *offset* argument and the number of pixels in *srcBuffer* specify a region of contiguous pixels in *destBuffer* to perform the compositing in.

Because single image strategies accept a sparse image as its input and return a sparse image as its output, their most common compositing operation is to composite two sparse images together. Compositing together two sparse images allows the composition to skip over inactive pixels in both images. However, because the compositing cannot be done in place, the results must be written to a third sparse image, which results in extra memory allocation and copying.

```
void icetCompressedCompressedComposite(
                              const IceTSparseImage  front_buffer,
                              const IceTSparseImage  back_buffer,
                              IceTSparseImage        dest_buffer  );
```

icetCompressedCompressedComposite takes two images, composites them, and places the results in *dest_buffer*. Unlike the previously mentioned forms of compositing, the blending order is not determined by a flag. Rather, the first image argument, *front_buffer*, is the image always considered closer to the viewer.

Communications

IceT provides an abstract communication layer, which is described in detail in Chapter 7. A handle to a communicator is stored in the current context. To make using the communicator easier, a set of convenience functions described next is available in the `IceTDevCommunication.h` include file. All of these functions are based off of those found in the MPI standard. For documentation, see that for the corresponding MPI function. Data types, however, are specified as one of the following IceT data types: **ICET_BOOLEAN**, **ICET_BYTE**, **ICET_SHORT**, **ICET_INT**, **ICET_SIZE_TYPE**, **ICET_FLOAT**, or **ICET_DOUBLE**. There is also an **ICET_IN_PLACE_COLLECT** identifier that takes the place of **MPI_IN_PLACE** for the gather functions (**icetCommGather**, **icetCommGatherv**, and **icetCommAllgather**).

Note that each function is missing an argument specifying the communicator. These functions just grab the current context's communicator. Also unlike MPI, these functions do not return error codes. It is assumed that errors are fatal. Some functions, like **icetCommSize**, **icetCommRank**, and **icetCommWaitany** use the function return value instead of passing a value back in a pointer argument.

```
struct IceTCommunicatorStruct *icetCommDuplicate(void);

void icetCommBarrier(void);

void icetCommSend(  const void *  buf,
                    int           count,
                    IceTEnum      datatype,
                    int           dest,
                    int           tag  );
```

```
void icetCommRecv( void *    buf,
                   int       count,
                   IceTEnum  datatype,
                   int       src,
                   int       tag  );

void icetCommSendrecv( const void *  sendbuf,
                       int           sendcount,
                       IceTEnum      sendtype,
                       int           dest,
                       int           sendtag,
                       void *        recvbuf,
                       int           recvcount,
                       IceTEnum      recvtype,
                       int           src,
                       int           recvtag  );

void icetCommGather( const void *  sendbuf,
                     int           sendcount,
                     IceTEnum      datatype,
                     void *        recvbuf,
                     int           root  );

void icetCommGatherv( const void *          sendbuf,
                      int                   sendcount,
                      IceTEnum              datatype,
                      void *                recvbuf,
                      const IceTSizeType *  recvcounts,
                      const IceTSizeType *  recvoffsets,
                      int                   root  );

void icetCommAllgather( const void *  sendbuf,
                        int           sendcount,
                        IceTEnum      type,
                        void *        recvbuf  );

void icetCommAlltoall( const void *  sendbuf,
                       int           sendcount,
                       IceTEnum      type,
                       void *        recvbuf  );

IceTCommRequest icetCommIsend( const void *  buf,
                               int           count,
                               IceTEnum      datatype,
                               int           dest,
                               int           tag  );
```

```
IceTCommRequest icetCommIrecv( void *    buf,
                               int       count,
                               IceTEnum  datatype,
                               int       src,
                               int       tag  );

void icetCommWait( IceTCommRequest *  request  );

void icetCommWaitany(
                     int                  count,
                     IceTCommRequest *    array_of_requests  );

int icetCommSize(void);

int icetCommRank(void);
```

In addition to these MPI-like functions, `IceTDevCommunication.h` provides a couple of helper functions for finding ranks in process groups. A group in IceT is simply represented by an integer array that maps (via the index) the rank in the group to the rank of the same process in the current context's communicator. An example of such a group is passed to the single image strategy compose function as demonstrated at the beginning of this chapter.

```
int icetFindRankInGroup( const int *   group,
                         IceTSizeType  group_size,
                         int           rank_to_find  );

int icetFindMyRankInGroup( const int *   group,
                           IceTSizeType  group_size  );
```

These functions provide the reverse mapping of a rank from the context's communicator to a rank in the group. The first form, **icetFindRankInGroup**, takes a *group* (and its *group_size*) and a *rank* and returns the index in *group* that contains *rank*. If *rank* is not in *group*, -1 is returned. **icetFindMyRankInGroup** is similar except that it returns the index for the local rank. Calling **icetFindMyRankInGroup** is equivalent to calling **icetFindRankInGroup** with *rank* set to the value in the state variable **ICET_RANK**.

TRANSFERRING IMAGES

Although the **IceTImage** and **IceTSparseImage** types are opaque, IceT provides a mechanism to transfer the data. A pair of functions allow you to package the data into a buffer (provided for you) and then unpackage a buffer back into an image object. The first of these functions is **icetImagePackageForSend** for **IceTImage** or **icetSparseImagePackageForSend** for **IceTSparseImage**.

```
void icetImagePackageForSend( IceTImage     image,
                              IceTVoid **   buffer,
                              IceTSizeType * size  );

void icetSparseImagePackageForSend( IceTSparseImage  image,
                                    IceTVoid **      buffer,
                                    IceTSizeType *   size  );
```

Both of these functions behave identically. They return a pointer in *buffer* to a block of raw data that can be sent opaquely via the communications functions. The length of this buffer in bytes is returned in *size*. When sending this data, send it as **ICET_BYTE** type data of *size* length. In the following example, an image is compressed and then its data is sent to another process.

```
IceTSparseImage src_sparse_image;
IceTVoid *package_buffer;
IceTSizeType package_size;

src_sparse_image = icetGetStateBufferSparseImage(MYCOMPOSITE_SRC_SPARSE_IMAGE,
                                                 max_width, max_height);
icetSparseImagePackageForSend(src_sparse_image, &package_buffer, &package_size);
icetCommSend(package_buffer, package_size, ICET_BYTE, dest_rank,
             MYCOMPOSITE_FOO_TAG);
```

The companion to each package function is the unpackage function. These are **icetImageUnpackageFromReceive** for **IceTImage** and **icetSparseImageUnpackageFromReceive** for **IceTSparseImage**.

```
IceTImage icetImageUnpackageFromReceive( IceTVoid *  buffer  );

IceTSparseImage icetSparseImageUnpackageFromReceive(
                                                IceTVoid *  buffer  );
```

These functions take a buffer containing the same data provided by a package command, create an image object, and return that object. Note that the buffer provided becomes part of the image. If that buffer is destroyed the image reverts to an undefined state.

This leads to a minor complication when receiving images. The receiver must allocate a raw buffer of the appropriate size and then leave it available while the image is still in use. This is best done by creating a state buffer (described in the Memory Management section starting on page 79). The size necessary for these buffers is determined with the buffer size functions, repeated here for reference.

```
IceTSizeType icetImageBufferSize( IceTSizeType  width,
                                  IceTSizeType  height  );

IceTSizeType icetSparseImageBufferSize( IceTSizeType  width,
                                        IceTSizeType  height  );
```

As a companion to the previous example, here a sparse image is received from another process and then composited into a locally held `image`.

```
IceTSparseImage dest_sparse_image;
IceTVoid *package_buffer;
IceTSizeType max_package_size;

max_package_size = icetSparseImageBufferSize(max_width, max_height);
package_buffer = icetGetStateBuffer(MYCOMPOSITE_DEST_SPARSE_IMAGE,
                                    max_package_size);
icetCommRecv(package_buffer, max_package_size, ICET_BYTE, src_rank,
             MYCOMPOSITE_FOO_TAG);
dest_sparse_image = icetSparseImageUnpackageFromReceive(package_buffer);
icetCompressedComposite(image, dest_sparse_image, ICET_FALSE);
```

HELPER COMMUNICATION FUNCTIONS

`common.h` (found in the strategies directory) contains some helper functions that implement common communication patterns. They may be helpful in implementing your strategy.

```
void icetRenderTransferFullImages(
                        IceTImage         image,
                        IceTVoid *        inSparseImageBuffer,
                        IceTSparseImage   outSparseImage,
                        IceTInt *         tile_image_dest  );
```

icetRenderTransferFullImages renders all the tiles that are specified in the **ICET_CONTAINED_TILES_LIST** state array and sends them to the processors with ranks specified in *tile_image_dest*. This function is guaranteed not to deadlock so long as all processes call it. The function uses only memory given with the buffer arguments, and will make its best efforts to get the graphics and network hardware to run in parallel.

image is an image object big enough to hold color and/or depth values that is **ICET_TILE_MAX_WIDTH** $\times$ **ICET_TILE_MAX_HEIGHT** big. *outSparseImage* is likewise a sparse image that big. *inSparseImageBuffer* is a buffer big enough to hold sparse color and depth information for an image that is **ICET_TILE_MAX_WIDTH** $\times$ **ICET_TILE_MAX_HEIGHT** big. The size for *inSparseImageBuffer* can be determined with the **icetSparseImageBufferSize** function in `IceTDevImage.h`. *tile_image_dest* is an array where if tile t is in **ICET_CONTAINED_TILES_LIST**, then the rendered image for tile t is sent to *tile_image_dest*$[t]$.

`common.h` also contains a similar function that does the same thing except that images are read back and returned as sparse images.

```
void icetRenderTransferSparseImages(
                          IceTSparseImage    compositeImage1,
                          IceTSparseImage    compositeImage2,
                          IceTVoid *         inSparseImageBuffer,
                          IceTSparseImage    outSparseImage,
                          IceTInt *          tile_image_dest,
                          IceTSparseImage *  resultImage  );
```

compositeImage1, *compositeImage2*, and *outSparseImage* are sparse image objects big enough to hold color and/or depth values that is **ICET_TILE_MAX_WIDTH** × **ICET_TILE_MAX_HEIGHT** big. *inImageBuffer* is a buffer big enough to hold sparse color and depth information for an image that is **ICET_TILE_MAX_WIDTH** × **ICET_TILE_MAX_HEIGHT** as determined by **icetSparseImageBufferSize**. *tile_image_dest* is an array where if tile t is in **ICET_CONTAINED_TILES_LIST**, then the rendered image for tile t is sent to *tile_image_dest*$[t]$. *resultImage* will be set to the image with the final results. It will point to either compositeImage1 or compositeImage2 depending on which buffer the result happened to end in.

There is also a more general form for transferring images or other large data blocks.

```
typedef IceTVoid *(*IceTGenerateData)(  IceTInt         id,
                                        IceTInt         dest,
                                        IceTSizeType *  size  );

typedef void (*IceTHandleData)(  void *   buffer,
                                 IceTInt  src  );

void icetSendRecvLargeMessages(
                          IceTInt             numMessagesSending,
                          const IceTInt *     messageDestinations,
                          IceTBoolean         messagesInOrder,
                          IceTGenerateData    generateDataFunc,
                          IceTHandleData      handleDataFunc,
                          IceTVoid *          incomingBuffer,
                          IceTSizeType        bufferSize);
```

icetSendRecvLargeMessages is similar to **icetRenderTransferFullImages** except that it works with generic data, data generators, and data handlers. It takes a count of a number of messages to be sent and an array of ranks to send to. Two callbacks are required. One generates the data (so large data may be generated JIT to save memory) and the other handles incoming data. The generate callback is run right before the data it returns is sent to a particular destination. This callback will not be called again until the memory it returned is no longer in use, so the memory may be reused. As large messages come in, the handle callback is called. As an optimization, if a process sends to itself, then that will be the first message created. This gives the callback an opportunity to build its local data while waiting for incoming data.

numMessagesSending is a count of the number of large messages this processor is sending out. *messageDestinations* is an array of size *numMessagesSending* that contains the ranks of message destinations. *generateDataFunc* is a callback function that generates messages. The function is given the index in *messageDestinations* and the rank of the destination as arguments. The data of the message and the size of the message (in bytes) are returned. The *generateDataFunc* will not be called again until the returned data is no longer in use. Thus the data may be reused. *handleDataFunc* is a callback function that processes messages. The function is given the data buffer and the rank of the process that sent it. The callback should completely finish its use of the buffer before returning. *incomingBuffer* is a buffer to use for incoming messages. *bufferSize* is the maximum size of a message.

Invoking Single-Image Compositing

Some of the existing IceT strategies internally use a more traditional single-image strategy to perform some of their work. Your multi-tile strategy might also benefit from leveraging these algorithms. To perform the composite of a single image amongst a group of processes, use the **icetSingleImageCompose** function defined in `common.h` in the strategies directory of IceT source code.

```
void icetSingleImageCompose(  const IceTInt *       compose_group,
                              IceTInt               group_size,
                              IceTInt               image_dest,
                              IceTSparseImage       input_image,
                              IceTSparseImage *     result_image,
                              IceTSizeType *        piece_offset  );
```

icetSingleImageCompose performs an image composition using the single-image strategy set by **icetSingleImageStrategy**. Rather than perform the composition on all the processes in the communicator, it performs them on a subset with arbitrary ordering. (Note that ordering matters when doing alpha blending as opposed to the z-buffer operation.) *compose_group* is the mapping of processes from the communicator ranks to the "group" ranks. The size of the groups (and the length of the *compose_group* array) is specified by *group_size*.

The *image_dest* argument provides a hint to the single image compositing algorithm where you plan to collect the resulting image data (usually with the **icetSingleImageCollect** function described next). The composed image evenutally will end up in the processor with rank *compose_group*[*image_dest*]. The compositing algorithm may use this hint to favor moving pixels to the indicated process.

input_image should contain the partial input image to be composited. (Of course, each process should have its own partial image. All processes should provide images of identical dimensions.) The contents of this buffer may be changed. *result_image* will be set to point to a composited image. In general, this is a partial image, so its size will likely be smaller than the original *input_image*. The location of the resulting piece is returned in *piece_offset*.

icetSingleImageCompose will generally return with the composited image partitioned amongst the processes in the group. If the **ICET_COLLECT_IMAGES** is on, then these peices must be collected to the display process. This is most easily done with the **icetSingleImageCollect** function.

```
void icetSingleImageCollect(
                        const IceTSparseImage  input_image,
                        IceTInt                dest,
                        IceTSizeType *         piece_offset,
                        IceTSparseImage *      result_image  );
```

icetSingleImageCollect collects image partitions distributed amongst processes. It is particularly useful after a call to **icetSingleImageCompose**. Unlike **icetSingleImageCompose**, however, this function must be called on all processes, not just those in a group. Processes that have no piece of the image should pass 0 for *piece_offset* and a null or other zero-size image for *input_image*.

The argument *input_image* contains the composited image partition returned from **icetSingleImageCompose**. *dest* is the rank of the process to where the image should be collected. Be aware that this is generally a different value than the *image_dest* parameter of **icetSingleImageCompose**. *piece_offset* is the offset to the start of the valid pixels. This is the same value as that returned from **icetSingleImageCompose**. *result_image* is an allocated image in which to place the uncompressed results of the collection.

If the **ICET_COLLECT_IMAGES** option is off, then the image collection, which can be one of the most time consuming parts of image compositing, can be skipped. If the image collection is skipped, then each process should set the values of **ICET_VALID_PIXELS_TILE**, **ICET_VALID_PIXELS_OFFSET**, and **ICET_VALID_PIXELS_NUM** to the tile, offset, and size, respectively, of the image piece returned by the local process. Setting these variables is unnecessary when images are collected to the display processes. It is valid to collect images even when **ICET_COLLECT_IMAGES** is off, but an error to not collect the images when this option is on.

The following is a very simple example of compositing the image on tile 0 and providing the result on the process displaying that tile. If ordered compositing is enabled, then the order is respected. This is similar to the sequential strategy except that only the first tile is composited.

```
#define COMPOSE_TILE_0_INPUT_IMAGE_BUFFER       ICET_STRATEGY_BUFFER_0
#define COMPOSE_TILE_0_OUTPUT_IMAGE_BUFFER      ICET_STRATEGY_BUFFER_1
#define COMPOSE_TILE_0_COMPOSE_GROUP            ICET_STRATEGY_BUFFER_2

IceTImage ComposeTile0(void)
{
  IceTInt max_width;
  IceTInt max_height;
  IceTInt rank;
  IceTInt num_proc;
  const IceTInt *display_nodes;
```

```
IceTInt image_dest;
IceTBoolean ordered_composite;
IceTSparseImage input_image;
IceTSparseImage composited_image;
IceTInt piece_offset;
IceTInt *compose_group;

icetGetIntegerv(ICET_NUM_TILES, &num_tiles);
icetGetIntegerv(ICET_TILE_MAX_WIDTH, &max_width);
icetGetIntegerv(ICET_TILE_MAX_HEIGHT, &max_height);
icetGetIntegerv(ICET_RANK, &rank);
icetGetIntegerv(ICET_NUM_PROCESSES, &num_proc);
display_nodes = icetUnsafeStateGetInteger(ICET_DISPLAY_NODES);
ordered_composite = icetIsEnabled(ICET_ORDERED_COMPOSITE);

input_image = icetGetStateBufferSparseImage(COMPOSE_TILE_0_INPUT_IMAGE_BUFFER,
                                            max_width, max_height);
output_image = icetGetStateBufferImage(COMPOSE_TILE_0_OUTPUT_IMAGE_BUFFER,
                                       max_width, max_height);
compose_group = icetGetStateBuffer(COMPOSE_TILE_0_COMPOSE_GROUP,
                                   sizeof(IceTInt)*num_proc);

if (ordered_composite) {
  icetGetIntegerv(ICET_COMPOSITE_ORDER, compose_group);
} else {
  int i;
  for (i = 0; i < num_proc; i++) {
    compose_group[i] = i;
  }
}

/* Determine which node in compose_group is displaying tile 0. */
image_dest = icetFindRankInGroup(compose_group, num_proc, display_nodes[0]);
if (image_dest < 0) {
  icetRaiseError("Could not find display node in composite order.",
                 ICET_SANITY_CHECK_FAIL);
}

icetGetCompressedTileImage(0, input_image);
icetSingleImageCompose(compose_group,
                       num_proc,
                       image_dest,
                       input_image,
                       &composited_image,
                       &piece_offset);

if (icetIsEnabled(ICET_COLLECT_IMAGES)) {
```

```
    icetSingleImageCollect(composited_image,
                           display_nodes[0],
                           piece_offset,
                           output_image);
  } else {
    IceTSizeType piece_size = icetSparseImageGetNumPixels(composited_image);
    if (piece_size > 0) {
      /* If the image is not collected, then the background of the piece
         should be corrected if ICET_NEED_BACKGROUND_CORRECTION is true.
         The icetDecompressSubImageCorrectBackground takes care of all this
         during decompression for you.  See the following section on
         background collection for more information. */
      icetDecompressSubImageCorrectBackground(composited_image,
                                              piece_offset,
                                              output_image);
      icetStateSetInteger(ICET_VALID_PIXELS_TILE, 0);
      icetStateSetInteger(ICET_VALID_PIXELS_OFFSET, piece_offset);
      icetStateSetInteger(ICET_VALID_PIXELS_NUM, piece_size);
    } else {
      output_image = icetImageNull();
      icetStateSetInteger(ICET_VALID_PIXELS_TILE, -1);
      icetStateSetInteger(ICET_VALID_PIXELS_OFFSET, 0);
      icetStateSetInteger(ICET_VALID_PIXELS_NUM, 0);
    }
  }

  return output_image;
}
```

Background Correction

As described in the volume/transparent rendering section of Chapter 4 (starting on page 49), it is sometimes necessary for IceT to correct the background of composited images by blending the final image over the background color. This responsibility falls on the compositing strategy (the multi-tile version, not the single-tile substrategy). The rational is that most efficient compositing strategies partition images and distribute for concurrent blending, and so the strategy can most efficiently correct the background by blending the pieces in parallel before it is collected.

Thus, if you are writing a new strategy, you should make sure that the background is always corrected when necessary. When your strategy function is called, the state variable **ICET_NEED_BACKGROUND_CORRECTION** will be set to true if the background needs to be corrected (i.e. **ICET_CORRECT_COLORED_BACKGROUND** is true, **icetCompositeMode** is set to **ICET_COMPOSITE_MODE_BLEND**, and the background color is not $[0,0,0,0]$). The actual background color to mix is stored in **ICET_TRUE_BACKGROUND_COLOR** and **ICET_TRUE_BACKGROUND_**

COLOR_WORD. These background colors can be blended on a per-pixel bases using the **ICET_BLEND_FLOAT** or **ICET_BLEND_UBYTE** macros defined in `IceTDevImage.h`.

That said, there is generally no any cause for a strategy to directly query these state variables. IceT provides several helper functions that automatically handle the background correction for you.

First, if your strategy is using the **icetSingleImageCollect** function, described in the previous section on invoking a single-image strategy, this function will automatically correct the background for you. No further works needs to be done.

`IceTDevImage.h` also provides several functions specifically to correct background data. First are **icetDecompressImageCorrectBackground** and **icetDecompressSubImageCorrectBackground**. These are analagous to the **icetDecompressImage** and **icetDecompressSubImage** functions except that they perform background correction if necessary. Since strategies often read images as compressed images and must decompress them before returning, this is often the most convienient way to correct the background. An example of using one is given in the previous section on invoking a single-image strategy for the case when image collection is not necessary.

```
void icetDecompressImageCorrectBackground(
                        const IceTSparseImage  compressed_image,
                        IceTImage              image  );

void icetDecompressSubImageCorrectBackground(
                        const IceTSparseImage  compressed_image,
                        IceTSizeType           offset,
                        IceTImage              image  );
```

If your strategy has already decompressed its image, it can correct the background by simply passing it to the **icetImageCorrectBackground** function. If no background correction is necessary, the function does nothing.

```
void icetImageCorrectBackground( IceTImage  image  );
```

Some compositing strategies will do no work for tiles with no geometry projected into them. In this case, the strategy must still return an image filled with the background color. The best way to create such an image is to use **icetClearImageTrueBackground**.

```
void icetClearImageTrueBackground( IceTImage  image  );
```

Matrix Operations

IceT uses 4×4 homogeneous transformation matrices to represent projections from object space to world and clipping space. These matrices are stored in `IceTDouble` arrays with 16 values

and have a layout conforming with matrices in OpenGL. To simplify the common operations on these matrices, IceT consolidates several useful matrix functions identified by the prototypes in `IceTDevMatrix.h`.

First is macro named `ICET_MATRIX`. This macro takes a pointer to a matrix array and row and column indices and resolves to the appropriate value in the matrix. The `ICET_MATRIX` macro ensures that row-column indices are properly converted to array indices. The following example use prints the values of a matrix.

```
IceTDouble matrix[16];
IceTInt row;
IceTInt column;

/* Do stuff that fills matrix. */

for (row = 0; row < 4; row++) {
  for (column = 0; column < 4; column++) {
    printf("%8.2lf", ICET_MATRIX(matrix, row, column);
  }
  printf("\n");
}
```

`IceTDevMatrix.h` contains a couple of matrix multiply functions. The first form, takes two matrices and stores the result into a third. The following computes $A \times B$ and stores the result in C.

```
void icetMatrixMultiply(  IceTDouble *        C,
                          const IceTDouble *  A,
                          const IceTDouble *  B  );
```

The second form as performs $A \times B$, but stores the result back into matrix A rather than in a third matrix. This operation is similar to the **glMultMatrix** function.

```
void icetMatrixPostMultiply(  IceTDouble *        A,
                              const IceTDouble *  B  );
```

The **icetMatrixVectorMultiply** function multiplies 4×4 matrix A with four-component column vector v and stores the result in array out (which of course must be allocated to hold 4 values).

```
void icetMatrixVectorMultiply(  IceTDouble *        out,
                                const IceTDouble *  A,
                                const IceTDouble *  v  );
```

`IceTDevMatrix.h` also contains two functions for performing the dot product of vectors. One performs the dot product for 3-component vectors, the other for 4-component vectors.

```
IceTDouble icetDot3( const IceTDouble *  v1,
                     const IceTDouble *  v2  );
IceTDouble icetDot4( const IceTDouble *  v1,
                     const IceTDouble *  v2  );
```

The **icetMatrixCopy** function provides a simple way to copy the data from one matrix array to another.

```
void icetMatrixCopy( IceTDouble *       matrix_dest,
                     const IceTDouble *  matrix_src  );
```

`IceTDevMatrix.h` contains functions to quickly build standard transformation matrices: identity, scale, translate, rotate, orthographic projection, and frustum projection. These functions have the same calling specifications as the OpenGL counterparts, **glLoadIdentity**, **glScale**, **glTranslate**, **glRotate**, **glOrtho**, and **glFrustum**, respectively, except that they all have a *mat_out* argument to store the resulting matrix.

```
void icetMatrixIdentity( IceTDouble *  mat_out  );

void icetMatrixScale( IceTDouble    x,
                      IceTDouble    y,
                      IceTDouble    z,
                      IceTDouble *  mat_out  );

void icetMatrixTranslate( IceTDouble    x,
                          IceTDouble    y,
                          IceTDouble    z,
                          IceTDouble *  mat_out  );

void icetMatrixRotate( IceTDouble    angle,
                       IceTDouble    x,
                       IceTDouble    y,
                       IceTDouble    z,
                       IceTDouble *  mat_out  );

void icetMatrixOrtho( IceTDouble    left,
                      IceTDouble    right,
                      IceTDouble    bottom,
                      IceTDouble    top,
                      IceTDouble    znear,
                      IceTDouble    zfar,
                      IceTDouble *  mat_out  );
```

```
void icetMatrixFrustum( IceTDouble    left,
                        IceTDouble    right,
                        IceTDouble    bottom,
                        IceTDouble    top,
                        IceTDouble    znear,
                        IceTDouble    zfar,
                        IceTDouble *  mat_out  );
```

In addition, `IceTDevMatrix.h` also provides other forms for the scale, translate, and rotate functions that apply (i.e. multiply) the transformation to an existing matrix. Again, the transformation is post multiplied in the same manner as OpenGL.

```
void icetMatrixMultiplyScale( IceTDouble *  mat_out,
                              IceTDouble    x,
                              IceTDouble    y,
                              IceTDouble    z  );

void icetMatrixMultiplyTranslate( IceTDouble *  mat_out,
                                  IceTDouble    x,
                                  IceTDouble    y,
                                  IceTDouble    z  );

void icetMatrixMultiplyRotate( IceTDouble *  mat_out,
                               IceTDouble    angle,
                               IceTDouble    x,
                               IceTDouble    y,
                               IceTDouble    z  );
```

Finally, `IceTDevMatrix.h` provides functions for computing the inverse, transpose, and inverse transpose of a matrix. The functions that perform an inverse return an `IceTBoolean` that is **ICET_TRUE** if the operation was successful or **ICET_FALSE** if the input matrix has no inverse.

```
IceTBoolean icetMatrixInverse( const IceTDouble *  matrix_in,
                               IceTDouble *        matrix_out  );

void icetMatrixTranspose( const IceTDouble *  matrix_in,
                          IceTDouble *        matrix_out  );

IceTBoolean icetMatrixInverseTranspose(
                           const IceTDouble *  matrix_in,
                           IceTDouble *        matrix_out  );
```

Raising Diagnostics

IceT's diagnostics system, described in Chapter 3 starting on page 30, alerts the user of anomalous conditions. Your compositing strategies should also alert the user through this diagnostic mechanism.

Error and warning diagnostics can be raised with the **icetRaiseError** and **icetRaiseWarning** functions, respectively. These functions are defined in the `IceTDevDiagnostics.h` header file.

```
void icetRaiseError( const char * message,
                     IceTEnum     type );

void icetRaiseWarning( const char * message,
                       IceTEnum     type );
```

The *message* argument contains a descriptive string that will be presented to the user describing the error or warning condition. The *type* argument gives the class of the error or warning. It must be one of the following with the given meanings.

ICET_INVALID_VALUE	An inappropriate value has been passed to a function.
ICET_INVALID_OPERATION	An inappropriate function has been called.
ICET_OUT_OF_MEMORY	IceT has ran out of memory for buffer space.
ICET_BAD_CAST	A function has been passed a value of the wrong type.
ICET_INVALID_ENUM	A function has been passed an invalid constant.
ICET_SANITY_CHECK_FAIL	An internal error (or warning) has occurred.

Your strategy also has the option of raising debug statements. Unlike an error or warning, debug statements are often raised during normal operation. These messages are *not* intended for the end user. Rather, they are status messages that might help you track problems while debugging. The debug messages are only created when IceT is compiled in "Debug" mode (the **CMAKE_BUILD_TYPE** CMake variable is set to `Debug` when building IceT) and the **ICET_DIAG_DEBUG** flag is given to **icetDiagnostics**. With these two conditions met, basic debug status messages can be raised with the **icetRaiseDebug** function.

```
void icetRaiseDebug( const char * message );
```

Unlike **icetRaiseError** and **icetRaiseWarning**, **icetRaiseDebug** does not accept a type because no anomalous condition is being reported.

It is common to want to use debug messages to report the state of variables. To help you with this, IceT provides three convenience functions that accept a printf-formatted message and a number of arguments. The message and arguments are passed to the C `sprintf` function.

```
void icetRaiseDebug1( const char * message,
                                 arg1  );

void icetRaiseDebug2( const char * message,
                                 arg1,
                                 arg2  );

void icetRaiseDebug4( const char * message,
                                 arg1,
                                 arg2,
                                 arg3,
                                 arg4  );
```

In the documentation for these functions here, the type for *argi* is left out. The type of these arguments is determined by the printf-formatted *message*. (As you have probably guessed, these functions are actually macros that pass the arguments to sprintf.)

Chapter 7

Communicators

IceT implements an abstract communication layer. As we will see later in this chapter, this communication layer is a message passing interface based heavily on MPI.[1] As an end user to IceT, you need to know almost nothing about this communication layer. You need only to get a reference to an **IceTCommunicator** object. This object is opaque. You only need to get one, pass it to **icetCreateContext**, and then delete it. **icetCreateContext** will duplicate the communicator, so you need not worry about when you delete the context you created.

Most of the time you will use the built-in MPI implementation of the communicator, which is discussed in the first section. If necessary, you can write your own communicator, which is discussed in the following section.

MPI Communicators

Using the MPI implementation of a communicator, you simply include `IceTMPI.h` in your source and link IceTMPI into your own library or executable. The only function you need to use is **icetCreateMPICommunicator**.

```
IceTCommunicator icetCreateMPICommunicator(
                                        MPI_Comm mpi_comm );
```

Quite simply, **icetCreateMPICommunicator** converts an **MPI_Comm**, an MPI communicator, into an **IceTCommunicator**, an IceT communicator. **icetCreateMPICommunicator** duplicates the MPI communicator. Thus, you can delete the *mpi_comm* communicator as soon as **icetCreateMPICommunicator** exits. Furthermore, the returned **IceTCommunicator** will internally manage the MPI communicator it created.

Once created, the **IceTCommunicator** may be deleted with **icetDestroyMPICommunicator**.

```
void icetDestroyMPICommunicator( IceTCommunicator comm );
```

[1] In fact, the original implementation of IceT used MPI directly. The abstract layer was inserted later as a more-or-less cut-and-paste operation.

icetDestroyMPICommunicator will release all the resources used by *comm*. This includes the internal MPI communicator, which you do not have direct access to. *comm* will be invalid once you call **icetDestroyMPICommunicator**. However, you do not have to worry about any IceT context you have passed it to since they will have duplicated the communicator.

Using the MPI communicator is easy. First, you include the `IceTMPI.h` header.

```
#include <IceT.h>
#include <IceTMPI.h>
```

When you are ready to create an IceT context (usually during the initialization of your program), create the MPI-based communicator, use it to initialize the context, and then destroy the communicator.

```
icetComm = icetCreateMPICommunicator(MPI_COMM_WORLD);
icetContext = icetCreateContext(icetComm);
icetDestroyMPICommunicator(icetComm);
```

Once you have a context, you can use IceT as explained throughout this document. When you are ready, destroy the context as you normally would.

```
icetDestroyContext(icetContext);
```

Finally, do not forget to use the IceTMPI library when linking your executable or library.

A more detailed example of using the MPI communicator is in the Chapter 2 tutorial.

User Defined Communicators

Occasionally, it may be necessary to provide your own version of a parallel communicator. This may be because you are using a communication library other than MPI. It may also be because you wish to augment the behavior of MPI when it is used by IceT. To provide your own communicator, you need only to create an **IceTCommunicator** object. In previous sections we have discussed **IceTCommunicator** as an opaque type, and unless you are implementing your own you should treat it as such. If you are implementing an **IceTCommunicator**, you will see that it is simply a pointer to a structure containing references to several communication functions.

```
typedef struct IceTCommRequestStruct {
    IceTEnum magic_number;
    IceTVoid *internals;
```

```
} *IceTCommRequest;
#define ICET_COMM_REQUEST_NULL ((IceTCommRequest)NULL)

struct IceTCommunicatorStruct {
    struct IceTCommunicatorStruct *
         (*Duplicate)(struct IceTCommunicatorStruct *self);
    void (*Destroy)(struct IceTCommunicatorStruct *self);
    void (*Barrier)(struct IceTCommunicatorStruct *self);
    void (*Send)(struct IceTCommunicatorStruct *self,
                 const void *buf,
                 int count,
                 IceTEnum datatype,
                 int dest,
                 int tag);
    void (*Recv)(struct IceTCommunicatorStruct *self,
                 void *buf,
                 int count,
                 IceTEnum datatype,
                 int src,
                 int tag);

    void (*Sendrecv)(struct IceTCommunicatorStruct *self,
                     const void *sendbuf,
                     int sendcount,
                     IceTEnum sendtype,
                     int dest,
                     int sendtag,
                     void *recvbuf,
                     int recvcount,
                     IceTEnum recvtype,
                     int src,
                     int recvtag);
    void (*Gather)(struct IceTCommunicatorStruct *self,
                   const void *sendbuf,
                   int sendcount,
                   IceTEnum datatype,
                   void *recvbuf,
                   int root);
    void (*Gatherv)(struct IceTCommunicatorStruct *self,
                    const void *sendbuf,
                    int sendcount,
                    IceTEnum datatype,
                    void *recvbuf,
                    const int *recvcounts,
                    const int *recvoffsets,
                    int root);
    void (*Allgather)(struct IceTCommunicatorStruct *self,
```

```
                        const void *sendbuf,
                        int sendcount,
                        IceTEnum datatype,
                        void *recvbuf);
    void (*Alltoall)(struct IceTCommunicatorStruct *self,
                     const void *sendbuf,
                     int sendcount,
                     IceTEnum datatype,
                     void *recvbuf);

    IceTCommRequest (*Isend)(struct IceTCommunicatorStruct *self,
                             const void *buf,
                             int count,
                             IceTEnum datatype,
                             int dest,
                             int tag);
    IceTCommRequest (*Irecv)(struct IceTCommunicatorStruct *self,
                             void *buf,
                             int count,
                             IceTEnum datatype,
                             int src,
                             int tag);

    void (*Wait)(struct IceTCommunicatorStruct *self, IceTCommRequest *request);
    int  (*Waitany)(struct IceTCommunicatorStruct *self,
                    int count, IceTCommRequest *array_of_requests);

    int  (*Comm_size)(struct IceTCommunicatorStruct *self);
    int  (*Comm_rank)(struct IceTCommunicatorStruct *self);
    void *data;
};

typedef struct IceTCommunicatorStruct *IceTCommunicator;
```

To create a custom **`IceTCommunicator`** simply allocate the structure and fill in the function pointers. An implementation for a function that creates an IceT communicator might look like the following. In this example, the `my*` functions are implementations of the communication functions.

```
IceTCommunicator myCreateCommunicator(myCommType myComm)
{
  IceTCommunicator comm = malloc(sizeof(struct IceTCommunicatorStruct));

  comm->Duplicate = myDuplicate;
  comm->Destroy = myDestroy;
  comm->Send = mySend;
  /* And so on... */
```

```
  comm->data = malloc(sizeof(myComm))
  /* Making a duplicate here would be better. */
  memcpy(comm->data, myComm, sizeof(myComm));

  return comm;
}
```

The paired destruction function should probably just call the Destroy function of the communicator (or vice versa) to ensure that destroy works either way.

```
void myDestroyCommunicator(IceTCommunicator comm)
{
  comm->Destroy(comm);
}


static void myDestroy(IceTCommunicator self)
{
  myCommType *myComm = (myCommType *)self->data;
  /* Release resources of myComm. */
  free(myComm);
  free(self);
}
```

For a more concrete example of implementing an IceT communicator, see the IceT code for the MPI communicator.

Chapter 8

Transitioning from IceT 1.0 to IceT 2

In the transition from IceT version 1 to IceT version 2, one of the major goals was to make the core IceT library independent of OpenGL. All of IceT's abilities to interface with OpenGL are retained but isolated in a separate library.

This change and others necessitated changes in the IceT interface. This chapter provides simple instructions for transitioning existing code to the new IceT interface.

Header File Changes

Because previous versions of IceT were considered an OpenGL library, public header files were placed in a `GL` subdirectory. Previous code included `GL/ice-t.h` and often also included `GL/ice-t_mpi.h`.

```
#include <GL/ice-t.h>
#include <GL/ice-t_mpi.h>
```

These header files no longer exist. The header files are now no longer in the `GL` directory and have changed in case and spelling to `IceT.h` and `IceTMPI.h`. There is also a new header file called `IceTGL.h` that contains the specific OpenGL functionality. A straight transition to IceT 2 will require this header file as well.

```
#include <IceT.h>
#include <IceTGL.h>
#include <IceTMPI.h>
```

Basic Type Changes

IceT 1 used the basic types defined by OpenGL such as `GLint` and `GLfloat`. IceT now defines its own basic types such as `IceTInt` and `IceTFloat`, and these new types should be used in place of the OpenGL types with respect to data passed to and from IceT.

Function Name Changes

To make the OpenGL layer explicit, all functions in this layer are prefixed with `icetGL`. This along with some other minor implementation details require slight changes in existing code.

First, before any other `icetGL` functions are called, **`icetGLInitialize`** must be called. **`icetGLInitialize`** should be called right after the context is created:

```
icetCreateContext(comm);
icetGLInitialize();
```

Make certain to call **`icetGLInitialize`** for *each* context for which you use the OpenGL layer.

Apart from adding a function call for **`icetGLInitialize`**, there are only two function names that have changed: **`icetDrawFunc`** and **`icetDrawFrame`** have changed to **`icetGL-DrawCallback`** and **`icetGLDrawFrame`**, respectively.

Technically, a function called **`icetDrawFrame`** still exists, but its interface has changed (so you should get a compiler error if you try to use it) and its behavior now skips any OpenGL specific operations (specifically reading and adjusting OpenGL state).

The function **`icetInputOutputBuffers`** has also been removed. It has been replaced with the two functions **`icetSetColorFormat`** and **`icetSetDepthFormat`**, which basically set the input buffers. The output buffers are implicitly set to be the same as the input buffers, but if the **`ICET_COMPOSITE_ONE_BUFFER`** feature is enabled, then the depth buffer will be suppressed in the output if both the color and depth buffer are read as input. This is the only sensible case for the input and output buffers to differ.

There is also a new function named **`icetCompositeMode`** to explicitly set the composite operation (z-buffer or blending). Previous versions of IceT implicitly set the composite mode based on what type of image data was available. To ensure that IceT is behaving as the user expects, this is now set explicitly. You will get errors if you have picked a composite mode that cannot be implemented with the image data available.

Getting Image Data

The **`icetGetColorBuffer`** and **`icetGetDepthBuffer`** functions no longer exist. The underlying data storage for images has become more flexible, and this method of getting image data became insufficient.

Instead, **`icetGLDrawFrame`** now returns an **`IceTImage`** object. There is a suite of new functions available, described in Chapter 3, that allow you to get data from **`IceTImage`** objects.

The methods that get color and data information from image objects are **icetImageGetColorub** and **icetImageGetDepthf**, respectively.

Also note that IceT now uses floating point numbers for depth values. Previous versions of IceT used 32-bit integers. Although using these integers was fast, IceT had problems with identifying background pixels. Different graphics hardware used different values for the maximum depths. However, the OpenGL specification places the maximum depth value at 1.0 if using floating point values.

Miscellaneous Changes

The serial strategy has been renamed to the sequential strategy. This better reflects the actual operation of the strategy as image compositing is still performed in parallel. As such, the **ICET_STRATEGY_SERIAL** identifier has changed to **ICET_STRATEGY_SEQUENTIAL**.

Libraries

The names of the libraries have changed from icet and icet_mpi to IceTCore and IceTMPI, respectively. Additionally, there is also an IceTGL library that contains code for the OpenGL layer. The icet_strategies library no longer exists. Everything in this library has been merged into IceTCore.

CMake Configuration

The CMake configuration for CMake has changed a bit between CMake version 1 and 2. You no longer need a copy of FindIceT.cmake. The `FIND_PACKAGE(IceT)` still works, but relies only on the IceTConfig.cmake file generated by IceT.

The use file prevously defined in **ICET_USE_FILE** no longer exists. Instead, you will need to insert your own commands to specify the IceT header and library locations.

There is a new variable called **ICET_GL_LIBS** that specifies the libraries used for the OpenGL layer. The variables **ICET_CORE_LIBS** and **ICET_MPI_LIBS** still exist to specify the core and MPI layer libraries, respectively.

Here is some typical CMake script fragment for using IceT.

```
FIND_PACKAGE(IceT REQUIRED)
```

```
INCLUDE_DIRECTORIES(${ICET_INCLUDE_DIRS})

ADD_EXECUTABLE(myprog ${SRCS})

TARGET_LINK_LIBRARIES(myproc
  ${ICET_CORE_LIBS}
  ${ICET_GL_LIBS}
  ${ICET_MPI_LIBS}
  )
```

Chapter 9

Future Work

The majority of the development for IceT was finished by 2004. Since then, IceT has proven to be a stable and versatile library that is currently being used in several production applications. In 2011 several scalability tests were run and the compositing algorithms updated to better support petascale computing.

The following is a list of potential changes to IceT. As of this writing, none of these are currently under development. Rather, these are identified shortcomings of various degrees in IceT. These features will be handled on an as needed basis, assuming the need should arise.

Render aborts In interactive applications, it is often convenient to be able to abort a render that takes some time to finish. Aborting a render in the middle of a composite is tricky, because you need to make sure that everyone is aware of the abort and that all communication is correctly canceled. This could be partially implemented in IceT's communication layer, but all the strategies still have to be ready to quit once a communication is canceled due to an abort (or at the very least ignore it without crashing).

Multithreaded compositing IceT is specifically designed for distributed memory parallel computing. It is clear that current and future high-performance computers are built with nodes comprising many computing cores each. It may be necessary to increase the thread safety of IceT and implement hybrid distributed-shared memory parallel compositing algorithms.

Memory conservation IceT's compositing algorithms require multiple image buffers. To save allocation time and prevent memory fragmenting, IceT keeps image buffers around in its own memory pool. Memory-constrained applications may prefer that IceT releases this memory between frames and take the performance hit. IceT's compositing algorithms could also stand a pass at minimizing the memory they use.

Multi-tile vs. big image compositing IceT contains several special algorithms for compositing multi-tile images. However, these strategies have never been properly compared to the compositing of a single very large image. This comparison should be done, and if the single compositing is faster then perhaps make a new strategy to combine tiles to a single large image before compositing, then split back up at the end.

Matrix functions IceT contains several convenience functions for 4×4 matrix transformations (described in Chapter 6 starting on page 106). They are hidden under the assumption that

programs using IceT will have similar functionality from its own rendering system, but perhaps that is not always the case. Should these functions be exposed?

Chapter 10

Man Pages

In this chapter you will find a man page for each of the functions available in the IceT API.

NAME

icetAddTile – add a tile to the logical display.

SYNOPSIS

```
#include <IceT.h>

int icetAddTile( IceTInt       x,
                 IceTInt       y,
                 IceTSizeType  width,
                 IceTSizeType  height,
                 int           display_rank  );
```

DESCRIPTION

Adds a tile to the tiled display. Every process, whether actually displaying a tile or not, must declare the tiles in the display and which processes drive them with **icetResetTiles** and **icetAddTile**. Thus, each process calls **icetAddTile** once for each tile in the display, and all processes must declare them in the same order.

The parameters *x*, *y*, *width*, and *height* define the tile's viewport in the logical global display much in the same way **glViewport** declares a region in a physical display in OpenGL. IceT places no limits on the extents of the logical global display. That is, there are no limits on the values of *x* and *y*. They can extend as far as they want in both the positive and negative directions.

IceT will project its images onto the region of the logical global display that just covers all of the tiles. Therefore, shifting all the tiles in the logical global display by the same amount will have no real overall effect.

The *display_rank* parameter identifies the rank of the process that will be displaying the given tile. It is assumed that the output of the rendering window of the given process is projected onto the space in a tiled display given by *x*, *y*, *width*, and *height*. Each tile must have a valid rank (between 0 and **ICET_NUM_PROCESSES** $-$ 1). Furthermore, no process may be displaying more than one tile.

RETURN VALUE

Returns the index of the tile created or -1 if the tile could not be created.

ERRORS

ICET_INVALID_VALUE Raised if *display_rank* is not a valid process rank, if *display_rank* is already assigned to another tile, or if *width* or *height* is smaller than 1. If this error is raised, nothing is done and -1 is returned.

WARNINGS

None.

BUGS

All processes must specify the same tiles in the same order. IceT will assume this even though it is not explicitly detected or enforced.

COPYRIGHT

SEE ALSO

icetResetTiles, **icetPhysicalRenderSize**

NAME

icetBoundingBoxd, **icetBoundingBoxf** – set bounds of geometry

SYNOPSIS

```
#include <IceT.h>

void icetBoundingBoxd( IceTDouble x_min,
                       IceTDouble x_max,
                       IceTDouble y_min,
                       IceTDouble y_max,
                       IceTDouble z_min,
                       IceTDouble z_max );

void icetBoundingBoxf( IceTFloat x_min,
                       IceTFloat x_max,
                       IceTFloat y_min,
                       IceTFloat y_max,
                       IceTFloat z_min,
                       IceTFloat z_max );
```

DESCRIPTION

Establishes the bounds of the geometry as contained in an axis-aligned box with the given extents.

icetBoundingBoxd and **icetBoundingBoxf** are really just convience functions. They create an array of the 8 corner vertices and set the bounding vertices appropriately. See **icetBoundingVertices** for more information.

ERRORS

None.

WARNINGS

None.

BUGS

None known.

COPYRIGHT

SEE ALSO

icetBoundingVertices

NAME

icetBoundingVertices – set bounds of geometry.

SYNOPSIS

```
#include <IceT.h>

void icetBoundingVertices( IceTInt           size,
                           IceTEnum          type,
                           IceTSizeType      stride,
                           IceTSizeType      count,
                           const IceTVoid *  pointer  );
```

DESCRIPTION

icetBoundingVertices is used to tell IceT what the bounds of the geometry drawn by the callback registered with **icetDrawCallback** or **icetGLDrawCallback** are. The bounds are assumed to be the convex hull of the vertices given. The user should take care to make sure that the drawn geometry actually does fit within the convex hull, or the data may be culled in unexpected ways. IceT runs most efficiently when the bounds given are tight (match the actual volume of the data well) and when the number of vertices given is minimal.

The *size* parameter specifies the number of coordinates given for each vertex. Coordinates are given in X-Y-Z-W order. Any Y or Z coordinate not given (because *size* is less than 3) is assumed to be 0.0, and any W coordinate not given (because *size* is less than 4) is assumed to be 1.0.

The *type* parameter specifies in what data type the coordinates are given. Valid *type*s are **ICET_SHORT**, **ICET_INT**, **ICET_FLOAT**, and **ICET_DOUBLE**, which correspond to types `IceTShort`, `IceTInt`, `IceTFloat`, and `IceTDouble`, respectively.

The *stride* parameter specifies the offset between consecutive vertices in bytes. If *stride* is 0, the array is assumed to be tightly packed.

The *count* parameter specifies the number of vertices to set.

The *pointer* parameter is an array of vertices with the first vertex starting at the first byte.

If data replication is being used, each process in a data replication group should register the same bounding vertices that encompass the entire geometry. By default there is no data replication, so unless you call **icetDataReplicationGroup**, all process can have their own bounds.

ERRORS

ICET_INVALID_ENUM	Raised if *type* is not one of **ICET_SHORT**, **ICET_INT**, **ICET_FLOAT**, or **ICET_DOUBLE**.

WARNINGS

None.

BUGS

None known.

COPYRIGHT

SEE ALSO

icetBoundingBox, **icetDataReplicationGroup**, **icetDrawCallback**, **icetGLDrawCallback**

NAME

icetCompositeMode – set the type of operation used for compositing

SYNOPSIS

```
#include <IceT.h>

void icetCompositeMode( IceTEnum mode );
```

DESCRIPTION

Sets the composite mode used when combining images. IceT enables parallel rendering by allowing each process in your code to independently render images of partial geometry. These partial-geometry images are then "composited" to form a single image equivalent to if all the geometry were rendered by a single process.

IceT supports multiple operations that can be used to combine images. The operator you use should be equivalent to that used by your rendering system to resolve hidden surfaces or mix occluding geometry with that behind it.

The argument *mode* is one of the following enumerations:

ICET_COMPOSITE_MODE_Z_BUFFER Use the z-buffer hidden-surface removal operation. The compositing operation compares the distance of pixel fragments from the viewpoint and passes the fragment closest to the user. In order for this operation to work, images must have a depth buffer (set with **icetSetDepthFormat**).

ICET_COMPOSITE_MODE_BLEND Blend two fragments together using the standard over/under operator. in order for this operation to work, images must have a color buffer (set with **icetSetColorFormat**) that has an alpha channel and there must be *no* depth buffer (as the operation makes no sense with depth). Also, this mode will only work if **ICET_ORDERED_COMPOSITE** is enabled and the order is set with **icetCompositeOrder**.

The default compositing mode is **ICET_COMPOSITE_MODE_Z_BUFFER**. The current composite mode is stored in the **ICET_COMPOSITE_MODE** state variable.

ERRORS

ICET_INVALID_ENUM *mode* is not a valid composite mode.

WARNINGS

None.

BUGS

icetCompositeMode will let you set a mode even if it is incompatible with other current settings. Some settings will be checked during a call to **icetDrawFrame**. For example, if the image format (specified with **icetSetColorFormat** and **icetSetDepthFormat**) does not support the composite mode picked, you will get an error during the call to **icetDrawFrame**.

Other incompatibilities are also not checked. For example, if the composite mode is set to **ICET_COMPOSITE_MODE_BLEND**, IceT will happily use this operator even if **ICET_ORDERED_COMPOSITE** is not enabled. However, because order matters in the blend mode, you will probably get incorrect images if the compositing happens in arbitrary order.

COPYRIGHT

SEE ALSO

icetCompositeOrder, **icetSetColorFormat**, **icetSetDepthFormat**

NAME

icetCompositeOrder – specify the order in which images are composited

SYNOPSIS

```
#include <IceT.h>

void icetCompositeOrder( const IceTInt * process_ranks );
```

DESCRIPTION

If **ICET_ORDERED_COMPOSITE** is enabled and the current strategy supports ordered composition (verified with the **ICET_STRATEGY_SUPPORTS_ORDERING** state variable, then the order which images are composited is specified with **icetCompositeOrder**. If compositing is done with z-buffer comparisons (e.g. **icetCompositeMode** is called with **ICET_COMPOSITE_MODE_Z_BUFFER**), then the ordering does not matter, and **ICET_ORDERED_COMPOSITE** should probably be disabled. However, if compositing is done with color blending (e.g. **icetCompositeMode** is called with **ICET_COMPOSITE_MODE_BLEND**), then the order in which the images are composed can drastically change the output.

For ordered image compositing to work, the geometric objects rendered by processes must be arranged such that if the geometry of one process is "in front" of the geometry of another process for any camera ray, that ordering holds for all camera rays. It is the application's responsibility to ensure that such an ordering exists and to find that ordering. The easiest way to do this is to ensure that the geometry of each process falls cleanly into regions of a grid, octree, k-d tree, or similar structure.

Once the geometry order is determined for a particular rendering viewpoint, it is given to IceT in the form of an array of ranks. The parameter *process_ranks* should have exactly **ICET_NUM_PROCESSES** entries, each with a unique, valid process rank. The first process should have the geometry that is "in front" of all others, the next directly behind that, and so on. It should be noted that the application may actually impose only a partial order on the geometry, but that can easily be converted to the linear ordering required by IceT.

When ordering is on, it is accepted that **icetCompositeOrder** will be called in between every frame since the order of the geometry may change with the viewpoint.

If data replication is in effect (see **icetDataReplicationGroup**), all processes are still expected to be listed in *process_ranks*. Correct ordering can be achieved by ensuring that all processes in each group are listed in contiguous entries in *process_ranks*.

ERRORS

ICET_INVALID_VALUE	Not every entry in the parameter *process_ranks* was a unique, valid process rank.

WARNINGS

None.

BUGS

If an **ICET_INVALID_VALUE** error is raised, internal arrays pertaining to the ordering of images may not be restored properly. If such an error is raised, the function should be re-invoked with a valid ordering before preceding. Unpredictable results may occur otherwise.

COPYRIGHT

SEE ALSO

icetCompositeMode **icetStrategy**

NAME

icetCopyState – copy state machine of one context to another.

SYNOPSIS

```
#include <IceT.h>

void icetCopyState( IceTContext       dest,
                    const IceTContext src );
```

DESCRIPTION

The **icetCopyState** function replaces the state of *dest* with the current state of *src*. This function can be used to quickly duplicate a context.

The **IceTCommunicator** object associated with *dest* is *not* changed (nor can it ever be). Consequently, the following state values are not copied either, since they refer to process ids that are directly tied to the **IceTCommunicator** object: **ICET_RANK**, **ICET_NUM_PROCESSES**, **ICET_DATA_REPLICATION_GROUP**, **ICET_DATA_REPLICATION_GROUP_SIZE**, **ICET_COMPOSITE_ORDER**, and **ICET_PROCESS_ORDERS**. However, every other state parameter is copied.

ERRORS

None.

WARNINGS

None.

BUGS

The state is copied blindly. It is therefore possible to copy states that are invalid for a context's communicator. For example, a display rank may not refer to a valid process id.

NOTES

Behavior is undefined if *dest* or *src* has never been created or has already been destroyed.

COPYRIGHT

SEE ALSO

icetCreateContext, **icetGetContext**, **icetSetContext**

NAME

icetCreateContext – creates a new context.

SYNOPSIS

```
#include <IceT.h>

IceTContext icetCreateContext( IceTCommunicator comm );
```

DESCRIPTION

The **icetCreateContext** function creates a new IceT context, makes it current, and returns a handle to the new context. The handle returned is of type **IceTContext**. This is an opaque type that should not be handled directly, but rather simply passed to other IceT functions.

Like OpenGL, the IceT engine behaves like a large state machine. The parameters for engine operation is held in the current state. The entire state is encapsulated in a context. Each new context contains its own state.

It is therefore possible to change the entire current state of IceT by simply switching contexts. Switching contexts is much faster, and often more convenient, than trying to change many state parameters.

ERRORS

None.

WARNINGS

None.

BUGS

It may be tempting to use contexts to run different IceT operations on separate program threads. Although certainly possible, great care must be taken. First of all, all threads will share the same context. Second of all, IceT is not thread safe. Therefore, a multi-threaded program would have to run all IceT commands in 'critical sections' to ensure that the correct context is being used, and the methods execute safely in general.

NOTES

icetCreateContext duplicates the communicator *comm*. Thus, to avoid deadlocks on certain implementations (such as MPI), the user level program should call **icetCreateContext** on all processes with the same *comm* object at about the same time.

COPYRIGHT

SEE ALSO

icetDestroyContext, **icetGetContext**, **icetSetContext**, **icetCopyState**, **icetGet**

NAME

icetCreateMPICommunicator – Converts an MPI communicator to an IceT communicator.

SYNOPSIS

```
#include <IceTMPI.h>

IceTCommunicator icetCreateMPICommunicator(
                                          MPI_Comm  mpi_comm  );
```

DESCRIPTION

IceT requires a communicator in order to perform correctly. An application is free to build its own communicator, but many will simply prefer to use MPI, which is a well established parallel communication tool. Thus, IceT comes with an implementation of **IceTCommunicator** that uses the MPI communication layer underneath.

icetCreateMPICommunicator is used to create an **IceTCommunicator** that uses the *mpi_comm* MPI communication object. The resulting **IceTCommunicator** shares the same process group and process rank as the original **MPI_Comm** communicator.

mpi_comm is duplicated, which has two consiquences. First, all process in *mpi_comm*'s group may need to call **icetCreateMPICommunicator** in order for any of them to proceed (depending on the MPI implementation). Second, *mpi_comm* and the resulting **IceTCommunicator** are decoupled from each other. Communications in one cannot affect another. Also, one communicator may be destroyed without affecting the other.

RETURN VALUE

An **IceTCommunicator** with the same process group and rank as *mpi_comm*. The communicator may be destroyed with a call to **icetDestroyMPICommunicator**.

ERRORS

None.

WARNINGS

None.

BUGS

All MPI errors are ignored.

COPYRIGHT

SEE ALSO

`icetDestroyMPICommunicator`, **`icetCreateContext`**

NAME

icetDataReplicationGroup – define data replication.

SYNOPSIS

```
#include <IceT.h>

void icetDataReplicationGroup( IceTInt          size,
                               const IceTInt *  processes  );
```

DESCRIPTION

IceT has the ability to take advantage of geometric data that is replicated among processes. If a group of processes share the same geometry data, then IceT will split the region of the display that the data projects onto among the processes, thereby reducing the total amount of image composition work that needs to be done.

Each group can be declared by calling **icetDataReplicationGroup** and defining the group of processes that share the geometry with the local process. *size* indicates how many processes belong to the group, and *processes* is an array of ids of processes that belong to the group. Each process that belongs to a particular group must call **icetDataReplicationGroup** with the exact same list of processes in the same order.

You can alternately use **icetDataReplicationGroupColor** to select data replication groups.

By default, each process belongs to a group of size one containing just the local processes (i.e. there is no data replication).

ERRORS

ICET_INVALID_VALUE *processes* does not contain the local process rank.

WARNINGS

None.

BUGS

IceT assumes that **icetDataReplicationGroup** is called with the exact same parameters on all processes belonging to a given group. Likewise, IceT also assumes that all processes have called **icetBoundingVertices** or **icetBoundingBox** with the exact same parameters on all processes belonging to a given group. These requirements are not strictly enforced, but failing to do so may cause some of the geometry to not be rendered.

COPYRIGHT

SEE ALSO

icetDataReplicationGroupColor, **icetBoundingVertices**, **icetBoundingBox**

NAME

icetDataReplicationGroupColor – define data replication.

SYNOPSIS

```
#include <IceT.h>

void icetDataReplicationGroupColor( IceTInt color );
```

DESCRIPTION

IceT has the ability to take advantage of geometric data that is replicated among processes. If a group of processes share the same geometry data, then IceT will split the region of the display that the data projects onto among the processes, thereby reducing the total amount of image composition work that needs to be done.

Despite the name of the function, **icetDataReplicationGroupColor** has nothing to do the color of the data being replicated. Instead, *color* is used to mark the local process as part of a given group. When **icetDataReplicationGroupColor** is called, it finds all other processes that have the same color and builds a group based on this information.

icetDataReplicationGroupColor must be called on every processes before the function will return.

ERRORS

None.

WARNINGS

None.

BUGS

IceT assumes that **icetDataReplicationGroup** is called with the exact same parameters on all processes belonging to a given group. Likewise, IceT also assumes that all processes have called **icetBoundingVertices** or **icetBoundingBox** with the exact same parameters on all processes belonging to a given group. These requirements are not strictly enforced, but failing

to do so may cause some of the geometry to not be rendered.

COPYRIGHT

SEE ALSO

icetDataReplicationGroup, **icetBoundingVertices**, **icetBoundingBox**

NAME

icetDestroyContext – delete a context.

SYNOPSIS

```
#include <IceT.h>

void icetDestroyContext( IceTContext context );
```

DESCRIPTION

Frees the memory required to hold the state of *context* and removes *context* from existence.

ERRORS

None.

WARNINGS

None.

BUGS

icetDestroyContext will happily delete the current context for you, but subsequent calls to most other IceT functions will probably result in seg-faults unless you make another context current with **icetCreateContext** or **icetSetContext**. The most notable execptions are the functions with names matching **icet*Context**, which will work correctly without a proper current context.

NOTES

Behavior is undefined if *context* has never been created or has already been destroyed.

COPYRIGHT

SEE ALSO

icetCreateContext

NAME

icetDestroyMPICommunicator – deletes a MPI communicator

SYNOPSIS

```
#include <IceTMPI.h>

void icetDestroyMPICommunicator( IceTCommunicator comm );
```

DESCRIPTION

Destroys an **IceTCommunicator**. *comm* becomes invalid and any memory or MPI resources held by *comm* are freed.

Communicators are copied when attached to an IceT context, so destroying an **IceTCommunicator** used to create a context still in use is safe.

ERRORS

None.

WARNINGS

None.

BUGS

All MPI errors are ignored.

COPYRIGHT

SEE ALSO

icetCreateMPICommunicator

NAME

icetDiagnostics – change diagnostic reporting level.

SYNOPSIS

```
#include <IceT.h>

void icetDiagnostics( IceTBitField mask );
```

DESCRIPTION

Sets what diagnostic message are printed to standard output. The messages to be printed out are defined by *mask*. *mask* consists of flags that are OR-ed together. The valid flags are:

ICET_DIAG_OFF A zero flag used to indicate that no diagnostic messages are desired.

ICET_DIAG_ERRORS Print messages associated with anomalous conditions.

ICET_DIAG_WARNINGS Print messages associated with conditions that are unexpected or may lead to errors. Implicitly turns on **ICET_DIAG_ERRORS**.

ICET_DIAG_DEBUG Print frequent messages concerning the status of IceT. Implicitly turns on **ICET_DIAG_ERRORS** and **ICET_DIAG_WARNINGS**.

ICET_DIAG_ROOT_NODE Print messages only on the node with a process rank of 0. This is the default if neither **ICET_DIAG_ROOT_NODE** nor **ICET_DIAG_ALL_NODES** is set.

ICET_DIAG_ALL_NODES Print messages all every nodes.

ICET_DIAG_FULL Turn on all diagnostic messages on all nodes.

The default flags are **ICET_DIAG_ALL_NODES** | **ICET_DIAG_WARNINGS**.

ERRORS

None.

WARNINGS

None.

BUGS

None known.

COPYRIGHT

SEE ALSO

`icetGetError`

NAME

icetDrawCallback – set a callback for drawing.

SYNOPSIS

```
#include <IceT.h>

typedef void (*IceTDrawCallbackType)(
                            const IceTDouble *  projection_matrix,
                            const IceTDouble *  modelview_matrix,
                            const IceTFloat *   background_color,
                            const IceTInt *     readback_viewport,
                            IceTImage           result  )

void icetDrawCallback(  IceTDrawCallbackType  callback  );
```

DESCRIPTION

The **icetDrawCallback** function sets a callback that is used to draw the geometry from a given viewpoint. If you are using OpenGL, you should probably use the **icetGLDrawCallback** function and associated **icetGLDrawFrame**. These alternative functions automatically set up the OpenGL state and retreive OpenGL buffers.

callback should be a function that renders an image of the local geometry based on the provided transformation matrices and background color. IceT will call *callback* during a call to **icetDrawFrame** to create the images for compositing. *callback* will be called a minimum amount of times. It may be called once. If none of the geometry projects on the display, it may not be called at all. If rendering to a tiled display and the geometry projects on multiple tiles, it may be called many times. The code in *callback* should be prepared to be called an unpredictable amount of times. For example, it should not attept to increment a frame counter and it should leave the rendering system's state such that another view to the geometry may be rendered.

callback takes two projection matrices: *projection_matrix* and *modelview_matrix*. Each of these arguments is a 16-value array that represents a 4×4 transformation of homogeneous coordinates. The arrays store the matrices in column-major order. Thus, if the values in *projection_matrix* are $(p[0], p[1], ...p[15])$ and the values in *modelview_matrix* are $(m[0], m[1], ...m[15])$, then a vertex in object space is transformed into normalized screen coordi-

nates by the sequence of operations

$$\begin{bmatrix} p[0] & p[4] & p[8] & p[12] \\ p[1] & p[5] & p[9] & p[13] \\ p[2] & p[6] & p[10] & p[14] \\ p[3] & p[7] & p[11] & p[15] \end{bmatrix} \begin{bmatrix} m[0] & m[4] & m[8] & m[12] \\ m[1] & m[5] & m[9] & m[13] \\ m[2] & m[6] & m[10] & m[14] \\ m[3] & m[7] & m[11] & m[15] \end{bmatrix} \begin{bmatrix} v[0] \\ v[1] \\ v[2] \\ v[3] \end{bmatrix}$$

Normalized screen coordinates are such that everything projected onto the image has coordinates in the range $[-1,1]$. The x and y coordinates have to be shifted to get the corresponding pixel location. The normalized screen coordinates are projected to span the physical render size (see **icetPhysicalRenderSize**), which may differ from the size of any particular tile. Also, if you are outputting depth values, IceT expects values in the range $[0,1]$, so you will have to shift those as well.

Note that the *projection_matrix* passed to *callback* is liable to be different than that passed to **icetDrawFrame**. Make certain that *callback* uses the modified *projection_matrix* passed to it. *modelview_matrix* is the same as that passed to **icetDrawFrame**, but also passed along for convienient reference.

Any pixel that does not have geometry rendered to it should be set to the *background_color* passed to *callback*. Likewise, any transparent geometry should be blended against the *background_color*. Note that the *background_color* passed to *callback* is liable to be different than that passed to **icetDrawFrame**.

callback is given *result*, an image object allocated to the size of the physical render size (see **icetPhysicalRenderSize**). The dimensions of the image can be queried with **icetImageGetWidth** and **icetImageGetHeight**. Pixels can be put in *result* by getting the color and/or depth buffers using the **icetImageGetColor** and **icetImageGetDepth** functions. Anything written to these buffers is captured in the image object.

IceT passes *callback* an image sized to the physical render space to make indexing into it clearer and safer and to possibly render directly into the image buffers. That said, IceT might only be interested in a subregion of the data. To make your callback more efficient, IceT provides *readback_viewport* to specify the region of the image it will read. *readback_viewport* has four values. The first two values specify the x and y pixel location of the lower left corner of the region of interest. The last two values specify the width and height of the region of interest. The callback only has to write valid pixels for this region of the image. It is not an error to write values outside this region, but they will be completely ignored.

The *callback* function pointer is placed in the **ICET_DRAW_FUNCTION** state variable.

ERRORS

None.

WARNINGS

None.

BUGS

None known.

NOTES

callback is tightly coupled with the bounds set with **icetBoundingVertices** or **icetBoundingBox**. If the geometry drawn by *callback* is dynamic (changes from frame to frame), then the bounds may need to be changed as well. Incorrect bounds may cause the geometry to be culled in surprising ways.

COPYRIGHT

SEE ALSO

icetBoundingBox, **icetBoundingVertices**, **icetDrawFrame**, **icetPhysicalRenderSize**

NAME

icetDrawFrame – renders and composites a frame

SYNOPSIS

```
#include <IceT.h>

IceTImage icetDrawFrame( const IceTDouble *  projection_matrix,
                         const IceTDouble *  modelview_matrix,
                         const IceTFloat *   background_color  );
```

DESCRIPTION

Initiates a frame draw using the given transformation matrices (modelview and projection). If you are using OpenGL, you should probably use the **icetGLDrawFrame** function and associated **icetGLDrawCallback**.

Before IceT may render an image, the tiled display needs to be defined (using **icetAddTile**), the drawing function needs to be set (using **icetDrawCallback**), and composite strategy must be set (using **icetStrategy**). The single image sub-strategy may also optionally be set (using **icetSingleImageStrategy**).

All processes in the current IceT context must call **icetDrawFrame** for it to complete.

During compositing, IceT uses the given *projection_matrix* and *modelview_matrix*, as well as the bounds given in the last call to **icetBoundingBox** or **icetBoundingVertices**, to determine onto which pixels the local geometry projects. If the given matrices are not the same used in the rendering or the given bounds do not contain the geometry, IceT may clip the geometry in surprising ways. Furthermore, IceT will modify the *projection_matrix* for the drawing callback to change the projection onto (or in between) tiles. Thus, you should pass the desired *projection_matrix* to **icetDrawFrame** and then use the version passed to the drawing callback.

RETURN VALUE

On each display process (as defined by **icetAddTile**, **icetDrawFrame** returns an image of the fully composited image. The contents of the image are undefined for any non-display process.

If the **ICET_COMPOSITE_ONE_BUFFER** option is on and both a color and depth buffer is specified with **icetSetColorFormat** and **icetSetDepthFormat**, then the returned image

might be missing the depth buffer. The rational behind this option is that often both the color and depth buffer is necessary in order to composite the color buffer, but the composited depth buffer is not needed. In this case, the compositing might save some time by not transferring depth information at the latter stage of compositing.

The returned image uses memory buffers that will be reclaimed the next time IceT renders a frame. Do not use this image after the next call to **icetDrawFrame** (unless you have changed the IceT context).

ERRORS

ICET_INVALID_OPERATION	Raised if the **icetGLInitialize** has not been called or if the drawing callback has not been set. Also can be raised if **icetDrawFrame** is called recursively, probably from within the drawing callback.
ICET_OUT_OF_MEMORY	Not enough memory left to hold intermittent frame buffers and other temporary data.

icetDrawFrame may also indirectly raise an error if there is an issue with the strategy or callback.

WARNINGS

None.

BUGS

If compositing with color blending on, the image returned may have a black background instead of the *background_color* requested. This can be corrected by blending the returned image over the desired background. This will be done for you if the **ICET_CORRECT_COLORED_BACKGROUND** is on.

COPYRIGHT

SEE ALSO

icetAddTile, **icetBoundingBox**, **icetBoundingVertices**, **icetDrawCallback**, **icetGLDrawFrame**, **icetSingleImageStrategy**, **icetStrategy**

NAME

icetEnable, **icetDisable**– enable/disable an IceT feature.

SYNOPSIS

```
#include <IceT.h>

void icetEnable   ( IceTEnum pname );
void icetDisable  ( IceTEnum pname );
```

DESCRIPTION

The **icetEnable** and **icetDisable** functions turn various IceT features on and off. *pname* is a symbolic constant representing the feature to be turned on or off. Valid values for *pname* are:

ICET_COLLECT_IMAGES When this option is on (the default) images partitions are always collected to display processes. When this option is turned off, the strategy has the option of leaving images partitioned among processes. Each process containing part of a tile's image will return the entire buffer from **icetDrawFrame** or **icetGLDrawFrame** in an **IceTImage** object. However, only certain pixels will be valid. The state variables **ICET_VALID_PIXELS_TILE**, **ICET_VALID_PIXELS_OFFSET**, and **ICET_VALID_PIXELS_NUM** give which tile the pixels belong to and what range of pixels are valid.

ICET_COMPOSITE_ONE_BUFFER Turn this option on when performing z-buffer compositing of a color image and the only result you need is the color image itself (not the depth buffer). This is common if you are just creating an image and are not interested in doing depth queries. This option is on by default.

ICET_CORRECT_COLORED_BACKGROUND Colored backgrounds are problematic when performing color blended compositing in that the background color will be additively blended from each image. Enabling this flag will internally cause the color to be reset to black and then cause the color to be blended back into the resulting images. This flag is disabled by default.

ICET_FLOATING_VIEWPORT If enabled, the projection will be shifted such that the geometry will be rendered in one shot whenever possible, even if the geometry straddles up to four tiles. This flag is enabled by default.

ICET_INTERLACE_IMAGES If enabled, pixels in images (might be) shuffled to better load balance the compositing work. This flag is enabled by default.

ICET_ORDERED_COMPOSITE If enabled, the image composition will be performed in the order specified by the last call to **icetCompositeOrder**, assuming the current strategy (specified by a call to **icetStrategy**) supports ordered composition. Generally, you want to enable ordered compositing if doing color blending and disable if you are doing z-buffer comparisons. If enabled, you should call **icetCompositeOrder** between each frame to update the image order as camera angles change. This flag is disabled by default.

In addition, if you are using the OpenGL layer (i.e., have called **icetGLInitialize**), these options, defined in `IceTGL.h`, are also available.

ICET_GL_DISPLAY If enabled, the final, composited image for each tile is written back to the frame buffer before the return of **icetGLDrawFrame**. This flag is enabled by default.

ICET_GL_DISPLAY_COLORED_BACKGROUND If this and **ICET_GL_DISPLAY** are enabled, IceT uses OpenGL blending to ensure that all background is set to the correct color. This flag is disabled by default. This option does not affect the images returned from **icetGLDrawFrame**; it only affects the image in the OpenGL color buffer.

ICET_GL_DISPLAY_INFLATE If this and **ICET_GL_DISPLAY** are enabled and the renderable window is larger than the displayed tile (as determined by the current OpenGL viewport), then resample the image to fit within the renderable window before writing back to frame buffer. This flag is disabled by default. This option does not affect the images returned from **icetGLDrawFrame**; it only affects the image in the OpenGL color buffer. If this option is not enabled, then images are written at their natural size in the lower left corner of the window.

ICET_GL_DISPLAY_INFLATE_WITH_HARDWARE This option determines how images are inflated. When enabled (the default), images are inflated by creating a texture and allowing the hardware to inflate the image. When disabled, images are inflated on the CPU. This option has no effect unless both **ICET_GL_DISPLAY** and **ICET_GL_DISPLAY_INFLATE** are also enabled.

ERRORS

ICET_INVALID_VALUE If *pname* is not a feature to be enabled or disabled.

WARNINGS

None.

BUGS

The check for a valid *pname* is not thorough, and thus the **ICET_INVALID_VALUE** error may not always be raised.

COPYRIGHT

SEE ALSO

icetIsEnabled

NAME

icetGet – get an IceT state parameter

SYNOPSIS

```
#include <IceT.h>

void icetGetDoublev( IceTEnum     pname,
                     IceTDouble * params );

void icetGetFloatv( IceTEnum    pname,
                    IceTFloat * params );

void icetGetIntegerv( IceTEnum   pname,
                      IceTInt *  params );

void icetGetBooleanv( IceTEnum      pname,
                      IceTBoolean * params );

void icetGetEnumv( IceTEnum   pname,
                   IceTEnum * params );

void icetGetBitFieldv( IceTEnum       pname,
                       IceTBitField * params );

void icetGetPointerv( IceTEnum    pname,
                      IceTVoid ** params );
```

DESCRIPTION

Like OpenGL, the operation of IceT is defined by a large state machine. Also like OpenGL, the state parameters can be retrieved through the **icetGet** functions. Each function takes a symbolic constant, *pname*, which identifies the state parameter to retrieve. They also each take an array, *params*, which will be filled with the values in *pname*. It is the calling application's responsibility to ensure that *params* is big enough to hold all the data.

STATE PARAMETERS

The following list identifies valid values for *pname* and a description of the associated state parameter.

ICET_BACKGROUND_COLOR The color that IceT is currently assuming is the background color. It is an RGBA value that is stored as four floating point values. This value is set either to the last value passed to **icetDrawFrame**, the OpenGL background color if **icetGLDrawFrame** was called, or to black for color blending. (The correct background color is restored later.)

ICET_BACKGROUND_COLOR_WORD The same as **ICET_BACKGROUND_COLOR** except that each component is stored as 8-bit RGBA values and packed in a 4-byte integer. The idea is to rapidly fill the background of color buffers.

ICET_BLEND_TIME The total time, in seconds, spent in performing color blending of images during the last call to **icetDrawFrame** or **icetGLDrawFrame**. Stored as a double. An alias for this value is **ICET_COMPARE_TIME**.

ICET_BUFFER_READ_TIME The total time, in seconds, spent copying buffer data and reading from OpenGL buffers during the last call to **icetDrawFrame** or **icetGLDrawFrame**. Stored as a double.

ICET_BUFFER_WRITE_TIME The total time, in seconds, spent writing to OpenGL buffers during the last call to **icetGLDrawFrame**. Always set to 0.0 after a call to **icetDrawFrame**. Stored as a double.

ICET_BYTES_SENT The total number of bytes sent by the calling process for transferring image data during the last call to **icetDrawFrame** or **icetGLDrawFrame**. Stored as an integer.

ICET_COLLECT_TIME The total time spent in collecting image fragments to display processes during the last call to **icetDrawFrame** or **icetGLDrawFrame**.

ICET_COLOR_FORMAT The color format of images to be created by the rendering subsystem and composited by IceT. Use **icetSetColorFormat** to set the color format. Use **icetImageGetColorFormat** to safely get the color format for a particular image.

ICET_COMPARE_TIME The total time, in seconds, spent in performing Z comparisons of images during the last call to **icetDrawFrame** or **icetGLDrawFrame**. Stored as a double. An alias for this value is **ICET_BLEND_TIME**.

ICET_COMPOSITE_MODE The composite mode set by **icetCompositeMode**. A single entry stored as an `IceTEnum`.

ICET_COMPOSITE_ORDER The order in which images are to be composited if **ICET_ORDERED_COMPOSITE** is enabled and the current strategy supports ordered compositing. The parameter contains **ICET_NUM_PROCESSES** entries. The value of this parameter is set with **icetCompositeOrder**. If the element of index i in the array is set to j, then there are i images "on top" of the image generated by process j.

ICET_COMPOSITE_TIME The total time, in seconds, spent in compositing during the last call to **icetDrawFrame** or **icetGLDrawFrame**.

Equal to **ICET_TOTAL_DRAW_TIME** − **ICET_RENDER_TIME** − **ICET_BUFFER_READ_TIME** − **ICET_BUFFER_WRITE_TIME**. Stored as a double.

ICET_COMPRESS_TIME The total time, in seconds, spent in compressing image data using active pixel encoding during the last call to **icetDrawFrame** or **icetGLDrawFrame**. Stored as a double.

ICET_DATA_REPLICATION_GROUP An array of process ids. There are **ICET_DATA_REPLICATION_GROUP_SIZE** entries in the array. IceT assumes that all processes in the list will create the exact same image with their draw functions (set with **icetDrawCallback** or **icetGLDrawCallback**). The local process id (**ICET_RANK**) will be part of this list.

ICET_DATA_REPLICATION_GROUP_SIZE The length of the **ICET_DATA_REPLICATION_GROUP** array.

ICET_DEPTH_FORMAT The depth format of images to be created by the rendering subsystem and composited by IceT. Use **icetSetDepthFormat** to set the depth format. Use **icetImageGetDepthFormat** to safely get the depth format for a particular image.

ICET_DIAGNOSTIC_LEVEL The diagnostics flags set with **icetDiagnostics**.

ICET_DISPLAY_NODES An array of process ranks. The size of the array is equal to the number of tiles (**ICET_NUM_TILES**). The i^{th} entry is the rank of the process that is displaying the tile described by the i^{th} entry in **ICET_TILE_VIEWPORTS**.

ICET_DRAW_FUNCTION A pointer to the drawing callback function, as set by **icetDrawCallback**.

ICET_FRAME_COUNT The number of times **icetDrawFrame** or **icetGLDrawFrame** has been called for the current context.

ICET_GEOMETRY_BOUNDS An array of vertices whose convex hull bounds the drawn geometry. Set with **icetBoundingVertices** or **icetBoundingBox**. Each vertex has three coordinates and are tightly packed in the array. The size of the array is $3 \times$ **ICET_NUM_BOUNDING_VERTS**.

ICET_GLOBAL_VIEWPORT Defines a viewport in an infinite logical display that covers all tile viewports (listed in **ICET_TILE_VIEWPORTS**). The viewport, like an OpenGL viewport, is given as the integer four-tuple $\langle x, y, width, height \rangle$. x and y are placed at the leftmost and lowest position of all the tiles, and *width* and *height* are just big enough for the viewport to cover all tiles.

ICET_MAGIC_K The target k value used when compositing with the radix-k single image strategy.

ICET_MAX_IMAGE_SPLIT The target number of maximum image splits to be performed by compositing strategies.

ICET_NUM_BOUNDING_VERTS The number of bounding vertices listed in the **ICET_GEOMETRY_BOUNDS** parameter.

ICET_NUM_TILES The number of tiles in the defined display. Basically equal to the number of times **icetAddTile** was called after the last **icetResetTiles**.

ICET_NUM_PROCESSES The number of processes in the parallel job as given by the **IceTCommunicator** object associated with the current context.

ICET_PHYSICAL_RENDER_HEIGHT The height of the images generated by the rendering system. This is set to the OpenGL viewport height by **icetGLDrawFrame** or otherwise by explicitly setting it with **icetPhysicalRenderSize** or otherwise implicitly to the largest tile height specified with **icetAddTile**.

ICET_PHYSICAL_RENDER_WIDTH The width of the images generated by the rendering system. This is set to the OpenGL viewport width by **icetGLDrawFrame** or otherwise by explicitly setting it with **icetPhysicalRenderSize** or otherwise implicitly to the largest tile width specified with **icetAddTile**.

ICET_PROCESS_ORDERS Basically, the inverse of **ICET_COMPOSITE_ORDER**. The parameter contains **ICET_NUM_PROCESSES** entries. If the element of index i in the array is set to j, then there are j images "on top" of the image generated by process i.

ICET_RANK The rank of the process as given by the **IceTCommunicator** object associated with the current context.

ICET_RENDER_TIME The total time, in seconds, spent in the drawing callback during the last call to **icetDrawFrame** or **icetGLDrawFrame**. Stored as a double.

ICET_SINGLE_IMAGE_STRATEGY The single image sub-strategy set with **icetSingleImageStrategy**. Use **icetGetSingleImageStrategyName** to get a user-readable name for the single image strategy.

ICET_STRATEGY The strategy set with **icetStrategy**. Use **icetGetStrategyName** to get a user-readable name for the strategy.

ICET_STRATEGY_SUPPORTS_ORDERING Is true if and only if the current strategy supports ordered compositing.

ICET_TILE_DISPLAYED The index of the tile the local process is displaying. The index will correspond to the tile entry in the **ICET_DISPLAY_NODES** and **ICET_TILE_VIEWPORTS** arrays. If set to $0 <= i <$ **ICET_NUM_PROCESSES**, then the i^{th} entry of **ICET_DISPLAY_NODES** is equal to **ICET_RANK**. If the local process is not displaying any tile, then **ICET_TILE_DISPLAYED** is set to -1.

ICET_TILE_MAX_HEIGHT The maximum *height* of any tile.

ICET_TILE_MAX_WIDTH The maximum *width* of any tile.

ICET_TILE_VIEWPORTS A list of viewports in the logical global display defining the tiles. Each viewport is the four-tuple $\langle x, y, width, height \rangle$ defining the position and dimensions of a tile in pixels, much like a viewport is defined in OpenGL. The size of the array is $4 * $ **ICET_NUM_TILES**. The viewports are listed in the same order as the tiles were defined with **icetAddTile**.

ICET_TOTAL_DRAW_TIME Time spent in the last call to **icetDrawFrame** or **icetGL-DrawFrame**. This includes all the time to render, read back, compress, and composite images. Stored as a double.

ICET_VALID_PIXELS_NUM In conjunction with **ICET_VALID_PIXELS_OFFSET**, gives the range of valid pixels for the last image returned from **icetDrawFrame** or **icetGL-DrawFrame**. Given the arrays of pixels returned with the **icetImageGetColor** and **icetImageGetDepth** functions, the valid pixels start at the pixel indexed by **ICET_VALID_PIXELS_OFFSET** and continue for **ICET_VALID_PIXELS_NUM**. The tile to which these pixels belong are captured in the **ICET_VALID_PIXELS_TILE** state variable. If the last call to **icetDrawFrame** or **icetGLDrawFrame** did not return pixels for the local process, **ICET_VALID_PIXELS_NUM** is set to 0. This state variable is only useful when **ICET_COLLECT_IMAGES** is off. If on, it can be assumed that all display processes contain all pixels in the image (**ICET_VALID_PIXELS_NUM** is the number of pixels in the image), and all other processes have no pixel data.

ICET_VALID_PIXELS_OFFSET In conjunction with **ICET_VALID_PIXELS_NUM**, gives the range of valid pixels for the last image returned from **icetDrawFrame** or **icetGL-DrawFrame**. Given the arrays of pixels returned with the **icetImageGetColor** and **icetImageGetDepth** functions, the valid pixels start at the pixel indexed by **ICET_VALID_PIXELS_OFFSET** and continue for **ICET_VALID_PIXELS_NUM**. The tile to which these pixels belong are captured in the **ICET_VALID_PIXELS_TILE** state variable. This state variable is only useful when **ICET_COLLECT_IMAGES** is off. If on, it can be assumed that all display processes contain all pixels in the image (**ICET_VALID_PIXELS_OFFSET** is 0), and all other processes have no pixel data.

ICET_VALID_PIXELS_TILE Gives the tile for which the last image returned from **icetDrawFrame** or **icetGLDrawFrame** contains pixels. Each process has its own value. If the last call to **icetDrawFrame** or **icetGLDrawFrame** did not return pixels for the local process, then this state variable is set to -1. This state variable is only useful when **ICET_COLLECT_IMAGES** is off. If on, it can be assumed that all display processes have valid pixels for their respective display tiles, and all other processes have no pixel data.

In addition, if you are using the OpenGL layer (i.e., have called **icetGLInitialize**), these variables, defined in `IceTGL.h`, are also available.

ICET_GL_DRAW_FUNCTION A pointer to the OpenGL drawing callback function, as set by **icetGLDrawCallback**.

ICET_GL_READ_BUFFER The OpenGL buffer to read from (and write to). Set with **icetGLSetReadBuffer**.

ERRORS

ICET_BAD_CAST	The state parameter requested is of a type that cannot be cast to the output type.
ICET_INVALID_ENUM	*pname* is not a valid state parameter.

WARNINGS

None.

BUGS

None known.

NOTES

Not every state variable is documented here. There is a set of parameters used internally by IceT or are more appropriately retrieved with other functions such as **icetIsEnabled**.

COPYRIGHT

SEE ALSO

icetIsEnabled, **icetGetStrategyName**

NAME

icetGetContext – retrieves the current context

SYNOPSIS

```
#include <IceT.h>

IceTContext icetGetContext( void );
```

DESCRIPTION

The **icetGetContext** function retrieves the handle for the current context. This handle may be stored and set for later use with **icetSetContext** (assuming the context has not been since destroyed).

RETURN VALUE

A handle for the current context.

ERRORS

None.

WARNINGS

None.

BUGS

None known.

COPYRIGHT

SEE ALSO

icetSetContext, **icetCreateContext**, **icetDestroyContext**, **icetCopyState**

NAME

icetGetError – return the last error condition.

SYNOPSIS

```
#include <IceT.h>

GLenum icetGetError( void );
```

DESCRIPTION

Retrieves the first error or warning condition that occurred since the last call to **icetGetError** or since program startup, whichever happened last.

Once an error condition has been retrieved with **icetGetError**, the error condition is reset to no error and cannot be retrieved again.

RETURN VALUE

One of the following flags will be returned:

ICET_INVALID_VALUE	An inappropriate value has been passed to a function.
ICET_INVALID_OPERATION	An inappropriate function has been called.
ICET_OUT_OF_MEMORY	IceT has ran out of memory for buffer space.
ICET_BAD_CAST	A function has been passed a value of the wrong type.
ICET_INVALID_ENUM	A function has been passed an invalid constant.
ICET_SANITY_CHECK_FAIL	An internal error (or warning) has occurred.
ICET_NO_ERROR	No error has been raised since the last call to **icetGetError**.

BUGS

It is not possible to tell if the returned value was caused by an error or a warning.

NOTES

The error value is *not* context dependent.

COPYRIGHT

SEE ALSO

icetDiagnostics

NAME

icetGetSingleImageStrategyName – retrieve single image sub-strategy name.

SYNOPSIS

```
#include <IceT.h>

const char *icetGetSingleImageStrategyName( void );
```

DESCRIPTION

icetGetSingleImageStrategyName retrieves a human-readable name for the current single image sub-strategy.

RETURN VALUE

Returns a short, null terminated string identifying the single image strategy currently in effect. Helpful for printing out debugging or diagnostic statements.

ERRORS

None.

WARNINGS

None.

BUGS

None known.

NOTES

The string returned does *not* contain the identifier used in a C program. For example, if the current single image strategy is **ICET_SINGLE_IMAGE_STRATEGY_BSWAP**, **icetGetS-**

ingleImageStrategyName returns “Binary Swap,” not “ICET_SINGLE_IMAGE_STRATEGY_BSWAP.”

COPYRIGHT

SEE ALSO

icetGetStrategyName, **icetSingleImageStrategy**

NAME

icetGetStrategyName – retrieve strategy name.

SYNOPSIS

```
#include <IceT.h>

const char *icetGetStrategyName( void );
```

DESCRIPTION

icetGetStrategyName retrieves a human-readable name for the current strategy.

RETURN VALUE

Returns a short, null terminated string identifying the strategy currently in effect. Helpful for printing out debugging or diagnostic statements. If no strategy is set, `NULL` is returned.

ERRORS

ICET_INVALID_ENUM	A strategy was never set with **icetStrategy**.

WARNINGS

None.

BUGS

None known.

NOTES

The string returned does *not* contain the identifier used in a C program. For example, if the current strategy is **ICET_STRATEGY_REDUCE**, **icetGetStrategyName** returns "Reduce," not "ICET_STRATEGY_REDUCE."

COPYRIGHT

SEE ALSO

icetGetSingleImageStrategyName, **icetStrategy**

NAME

icetGLDrawCallback – set a callback for drawing with OpenGL.

SYNOPSIS

```
#include <IceTGL.h>

typedef void (* IceTGLDrawCallbackType)( void );

void icetGLDrawCallback( IceTGLDrawCallbackType callback );
```

DESCRIPTION

The **icetGLDrawCallback** function sets a callback that is used to draw the geometry from a given viewpoint. It will be implicitly called from within **icetGLDrawFrame**.

callback should be a function that issues appropriate OpenGL calls to draw geometry in the current OpenGL context. After *callback* is called, the image left in the frame buffer specified by **icetGLSetReadBuffer** will be read back for compositing.

callback should *not* modify the **GL_PROJECTION_MATRIX** as this would cause IceT to place image data in the wrong location in the tiled display and improperly cull geometry. It is acceptable to add transformations to **GL_MODELVIEW_MATRIX**, but the bounding vertices given with **icetBoundingVertices** or **icetBoundingBox** are assumed to already be transformed by any such changes to the modelview matrix. Also, **GL_MODELVIEW_MATRIX** must be restored before the draw function returns. Therefore, any changes to **GL_MODELVIEW_MATRIX** are to be done with care and should be surrounded by a pair of glPushMatrix and glPopMatrix functions.

It is also important that *callback* *not* attempt the change the clear color. In some composting modes, IceT needs to read, modify, and change the background color. These operations will be lost if *callback* changes the background color, and severe color blending artifacts may result.

IceT may call *callback* several times from within a call to **icetGLDrawFrame** or not at all if the current bounds lie outside the current viewpoint. This can have a subtle but important impact on the behavior of *callback*. For example, counting frames by incrementing a frame counter in *callback* is obviously wrong (although you could count how many times a render occurs). *callback* should also leave OpenGL in a state such that it will be correct for a subsequent run of *callback*. Any matrices or attributes pushed in *callback* should be popped before *callback* returns, and any state that is assumed to be true on entrance to *callback* should also be true on return.

The *callback* function pointer is placed in the **ICET_GL_DRAW_FUNCTION** state variable.

icetGLDrawCallback is similar to **icetDrawCallback**. The difference is that the callback set by **icetGLDrawCallback** is used by **icetGLDrawFrame** and the callback set by **icetDrawCallback** is used by **icetDrawFrame**.

ERRORS

ICET_INVALID_OPERATION Raised if the **icetGLInitialize** has not been called.

WARNINGS

None.

BUGS

None known.

NOTES

callback is tightly coupled with the bounds set with **icetBoundingVertices** or **icetBoundingBox**. If the geometry drawn by *callback* is dynamic (changes from frame to frame), then the bounds may need to be changed as well. Incorrect bounds may cause the geometry to be culled in surprising ways.

COPYRIGHT

SEE ALSO

icetBoundingBox, **icetBoundingVertices**, **icetDrawCallback**, **icetGLDrawFrame**

NAME

icetGLDrawFrame – renders and composites a frame using OpenGL

SYNOPSIS

```
#include <IceTGL.h>

void  icetGLDrawFrame( void );
```

DESCRIPTION

Initiates a frame draw using the current OpenGL transformation matrices (modelview and projection).

Before IceT may render an image, the tiled display needs to be defined (using **icetAddTile**), the drawing function needs to be set (using **icetGLDrawCallback**), and composite strategy must be set (using **icetStrategy**). The single image sub-strategy may also optionally be set (using **icetSingleImageStrategy**).

All processes in the current IceT context must call **icetGLDrawFrame** for it to complete.

The OpenGL projection matrix and modelview matrix must be set using **glLoadMatrix** or a number of other functions. Likewise, the OpenGL background color must be set with **glClearColor**. IceT will use the matrices to determine what pixels need to be rendered and composited. IceT will also modify the projection matrix to project onto (or in between) tiles.

If **ICET_GL_DISPLAY** is enabled, then the fully composited image is written back to the OpenGL framebuffer for display. It is the application's responsibility to synchronize the processes and swap front and back buffers. If the OpenGL background color is set to something other than black, **ICET_GL_DISPLAY_COLORED_BACKGROUND** should also be enabled. Displaying with **ICET_GL_DISPLAY_COLORED_BACKGROUND** disabled may be slightly faster (depending on graphics hardware) but can result in black rectangles in the background. If **ICET_GL_DISPLAY_INFLATE** is enabled and the size of the renderable window (determined by the current OpenGL viewport) is greater than that of the tile being displayed, then the image will first be "inflated" to the size of the actual display. This inflation will be assisted by texture hardware if **ICET_GL_DISPLAY_INFLATE_WITH_HARDWARE** is on. If **ICET_GL_DISPLAY_INFLATE** is disabled, the image is drawn at its original resolution at the lower left corner of the display.

The image remaining in the frame buffer is undefined if **ICET_GL_DISPLAY** is disabled or the process is not displaying a tile.

ERRORS

ICET_INVALID_OPERATION	Raised if the drawing callback has not been set. Also can be raised if **icetDrawFrame** is called recursively, probably from within the drawing callback.
ICET_OUT_OF_MEMORY	Not enough memory left to hold intermittent frame buffers and other temporary data.

icetGLDrawFrame may also indirectly raise an error if there is an issue with the strategy or callback.

WARNINGS

None.

BUGS

If compositing with color blending on, the image returned may have a black background instead of the *background_color* requested. This can be corrected by blending the returned image over the desired background. This will be done for you if the **ICET_CORRECT_COLORED_BACKGROUND** is on.

COPYRIGHT

SEE ALSO

icetAddTile, **icetBoundingBox**, **icetBoundingVertices**, **icetDrawFrame**, **icetGLDrawCallback**, **icetSingleImageStrategy**, **icetStrategy**

NAME

icetGLInitialize – initialize the IceT OpenGL layer

SYNOPSIS

```
#include <IceTGL.h>

void icetGLInitialize( void );
```

DESCRIPTION

Initializes the OpenGL layer of IceT. **icetGLInitialize** must be called before any other function starting with `icetGL` (except **icetGLIsInitialized**).

Management for the OpenGL layer is held in the state of the current IceT context. Thus, **icetGLInitialize** must be called once per IceT context. If you are using a context for rendering with OpenGL, it is recommended that you call **icetGLInitialize** immediately after calling **icetCreateContext**.

ERRORS

None.

WARNINGS

ICET_INVALID_OPERATION **icetGLInitialize** is called twice for the same context.

BUGS

None known.

COPYRIGHT

SEE ALSO

icetCreateContext, **icetGLIsInitialized**

NAME

icetGLIsInitialized – determine if the IceT OpenGL layer is initialized

SYNOPSIS

```
#include <IceTGL.h>

IceTBoolean icetGLIsInitialized( void );
```

DESCRIPTION

Used to determine whether **icetGLInitialize** was called for the current IceT context. If **icetGLIsInitialized** returns false, then rendering with the OpenGL layer will not work.

RETURN VALUE

Returns **ICET_TRUE** if the OpenGL layer has been initialized for the current context, **ICET_FALSE** otherwise.

ERRORS

None.

WARNINGS

None.

BUGS

None known.

COPYRIGHT

SEE ALSO

icetGLInitialize

NAME

icetGLSetReadBuffer – set OpenGL buffer for images

SYNOPSIS

```
#include <IceTGL.h>

void icetGLSetReadBuffer( GLenum mode );
```

DESCRIPTION

Set the OpenGL buffer from which to read images to composite. After the draw callback (specified by **icetGLDrawCallback**) returns, IceT grabs the rendered image from the OpenGL buffer specified by *mode*. This buffer is also used to write back fully composited images if the **ICET_GL_DISPLAY** option is on.

mode is an OpenGL value that specifies the buffer. It is passed to **glReadBuffer** and **glDrawBuffer**. Accepted values are **GL_FRONT**, **GL_BACK**, **GL_LEFT**, **GL_RIGHT**, **GL_FRONT_LEFT**, **GL_FRONT_RIGHT**, **GL_BACK_LEFT**, **GL_BACK_RIGHT**, and any of the **GL_AUX***i* identifiers.

The current read buffer used is stored in the **ICET_GL_READ_BUFFER** state variable. The default value is **GL_BACK**.

ERRORS

ICET_INVALID_OPERATION	**icetGLInitialize** has not been called for this IceT context.
ICET_INVALID_ENUM	*mode* is not a known OpenGL buffer identifier.

WARNINGS

None.

BUGS

The check of *mode* is perfunctory. It just checks *mode* against a list of known buffers. It does not check to see if the buffer actually exists or for any other buffers that might be defined in an

OpenGL extension.

COPYRIGHT

SEE ALSO

icetGLDrawCallback

NAME

icetImageCopyColor, **icetImageCopyDepth**– retrieve pixel data from image

SYNOPSIS

```
#include <IceT.h>

void  icetImageCopyColorub ( const IceTImage  image,
                             IceTUByte *      color_buffer,
                             IceTEnum         color_format  );

void  icetImageCopyColorf ( const IceTImage  image,
                            IceTFloat *      color_buffer,
                            IceTEnum         color_format  );

void  icetImageCopyDepthf ( const IceTImage  image,
                            IceTFloat *      depth_buffer,
                            IceTEnum         depth_format  );
```

DESCRIPTION

The **icetImageCopyColor** suite of functions retrieve color data from images and the **icetImageCopyDepth** functions retrieve depth data from images. Each function takes a pointer to an existing buffer that must be large enough to hold all pixels in the image. The data from the images is copied into these buffers, performing format conversions as necessary. Because data is copied into the provided buffer, subsequently changing values in the buffer has no effect on the image object (as opposed to the behavior of **icetImageGetColor** and **icetImageGetDepth**).

The pixel data is always tightly packed in horizontal major order. Color data that comprises tuples such as RGBA have the components for each pixel packed together in that order. The first entry in the array corresponds to the pixel in the lower left corner of the image. The next entry is immediately to the right of the first pixel, and so on. The dimensions of the array can be retrieved with the **icetImageGetWidth** and **icetImageGetHeight** functions.

Each of these functions provides a typed version of the image data array. They can only succeed if the type the request matches the type specified by the *color_format* or *depth_format* argument. It is an error, for example, to request unsigned byte color data for a floating point color format. Although specifying the format may be redundant (it could be implied by the type being retrieved), IceT requires it for completeness and to support possible future data formats.

Use **icetImageCopyColorub** to retrieve an array of 8-bit unsigned bytes. Using this function is only valid if *color_format* is **ICET_IMAGE_COLOR_RGBA_UBYTE**.

Use **icetImageCopyColorf** to retrieve an array of floating point color values. Using this function is only valid if *color_format* is **ICET_IMAGE_COLOR_RGBA_FLOAT**.

Use **icetImageGetDepthf** to retrieve an array of floating point depth values. Using this function is only valid if *depth_format* is **ICET_IMAGE_DEPTH_FLOAT**.

ERRORS

ICET_INVALID_OPERATION	The *image* object does not have a color or depth buffer from which to copy data.
ICET_INVALID_ENUM	The requested *color_format* or *depth_format* is incompatible with the type of the buffer.

WARNINGS

None.

BUGS

None known.

COPYRIGHT

SEE ALSO

icetImageGetColor, **icetImageGetDepth**

NAME

icetImageGetColor, **icetImageGetDepth**– retrieve pixel data buffer from image

SYNOPSIS

```
#include <IceT.h>

IceTUByte *  icetImageGetColorub  ( IceTImage  image );
IceTUInt *   icetImageGetColorui  ( IceTImage  image );
IceTFloat *  icetImageGetColorf   ( IceTImage  image );

IceTFloat *  icetImageGetDepthf   ( IceTImage  image );

const IceTUByte *  icetImageGetColorcub  ( const IceTImage  image );
const IceTUInt *   icetImageGetColorcui  ( const IceTImage  image );
const IceTFloat *  icetImageGetColorcf   ( const IceTImage  image );

const IceTFloat *  icetImageGetDepthcf   ( const IceTImage  image );
```

DESCRIPTION

The **icetImageGetColor** suite of functions retrieve color data from images and the **icetImageGetDepth** functions retrieve depth data from images. Each function returns a pointer to an internal buffer within the image. Writing to this data changes the data within the image object itself. Use the **icetImageGetColor** and **icetImageGetDepth** functions from within drawing callbacks to pass image data back to IceT.

The pixel data is always tightly packed in horizontal major order. Color data that comprises tuples such as RGBA have the components for each pixel packed together in that order. The first entry in the array corresponds to the pixel in the lower left corner of the image. The next entry is immediately to the right of the first pixel, and so on. The dimensions of the array can be retrieved with the **icetImageGetWidth** and **icetImageGetHeight** functions.

Each of these functions returns a typed version of the image data array. They can only succeed if the type the request matches the internal type of the array. It is an error, for example, to request unsigned byte color data when the image stores images as floating point colors. You can use the **icetImageGetColorFormat** and **icetImageGetDepthFormat** to retrieve the format for the internal data storage (which also implies the base data type). You can also use the **icetImageCopyColor** and **icetImageCopyDepth** functions to convert the image data to whatever format you like.

Use **icetImageGetColorub** to retrieve an array of 8-bit unsigned bytes. Using this func-

tion is only valid if the color format is **ICET_IMAGE_COLOR_RGBA_UBYTE**.

Use **icetImageGetColorui** to retrieve an array of 32-bit unsigned integers. Using this function is only valid if the color format is **ICET_IMAGE_COLOR_RGBA_UBYTE**. In this case, each 32-bit integer represents all four RGBA channels. Accessing each pixel's color values as a single 32-bit integer is often faster than accessing it as 4 independent 8-bit integers as most modern architectures can access 32-bit memory boundaries faster than independent 8-bit boundaries.

Use **icetImageGetColorf** to retrieve an array of floating point color values. Using this function is only valid if the color format is **ICET_IMAGE_COLOR_RGBA_FLOAT**.

Use **icetImageGetDepthf** to retrieve an array of floating point depth values. Using this function is only valid if the depth format is **ICET_IMAGE_DEPTH_FLOAT**.

RETURN VALUE

Returns an appropriately typed array pointing to the internal color or depth values stored in the image object. If there is an error, `NULL` is returned.

The memory returned should not be freed. It is managed internally by IceT.

ERRORS

ICET_INVALID_OPERATION	The internal color or depth format is incompatible with the type of array the function retrieves.

WARNINGS

None.

BUGS

None known.

NOTES

There is no mechanism to automatically determine the data type from the color or depth format enumeration (returned from **icetImageGetColorFormat** or **icetImageGetDepthFormat**). Instead, you must code internal logic to use an array of the appropriate type. The reasoning behind this decision is that the format encodes the data layout in addition to the data type, and your

code most understand the basic semantics of the data to do anything worthwhile with it. If you want to write code that is indifferent to the underlying format of the image, use the **icetImageCopyColor** and **icetImageCopyDepth** functions to copy the data to a known format.

COPYRIGHT

SEE ALSO

icetImageCopyColor, **icetImageCopyDepth**, **icetImageGetColorFormat**, **icetImageGetDepthFormat**

NAME

icetImageGetColorFormat, **icetImageGetDepthFormat**– get the format of image buffers

SYNOPSIS

```
#include <IceT.h>

IceTEnum icetImageGetColorFormat( const IceTImage image );
IceTEnum icetImageGetDepthFormat( const IceTImage image );
```

DESCRIPTION

icetImageGetColorFormat and **icetImageGetDepthFormat** return an entry in an enumeration that specifies the data format stored in the given *image*. This format determines which one of the **icetImageGetColor** or **icetImageGetDepth** functions to use and the form of the resulting data.

RETURN VALUE

icetImageGetColorFormat returns one of the following values with the associated meaning for the format of the stored color data.

ICET_IMAGE_COLOR_RGBA_UBYTE	Each entry is an RGBA color tuple. Each component is valued in the range from 0 to 255 and is stored as an 8-bit integer. The buffer will always be allocated on memory boundaries such that each color value can be treated as a single 32-bit integer.
ICET_IMAGE_COLOR_RGBA_FLOAT	Each entry is an RGBA color tuple. Each component is in the range from 0.0 to 1.0 and is stored as a 32-bit float.
ICET_IMAGE_COLOR_NONE	No color values are stored in the image.

icetImageGetDepthFormat returns one of the following values with the associated meaning for the format of the stored depth data.

ICET_IMAGE_DEPTH_FLOAT	Each entry is in the range from 0.0 (near plane) to 1.0 (far plane) and is stored as a 32-bit float.

ICET_IMAGE_DEPTH_NONE	No depth values are stored in the image.

ERRORS

None.

WARNINGS

None.

BUGS

None known.

COPYRIGHT

SEE ALSO

icetImageGetColor, **icetImageGetDepth**, **icetSetColorFormat**, **icetSetDepthFormat**

NAME

icetImageGetWidth, **icetImageGetHeight**, **icetImageGetNumPixels**– get dimensions of an image

SYNOPSIS

```
#include <IceT.h>

IceTSizeType  icetImageGetWidth      ( const IceTImage  image );
IceTSizeType  icetImageGetHeight     ( const IceTImage  image );
IceTSizeType  icetImageGetNumPixels  ( const IceTImage  image );
```

DESCRIPTION

icetImageGetWidth, **icetImageGetHeight**, and **icetImageGetNumPixels** allow you to query the size of an *image* with respect to the number of pixels. These functions define the buffer size returned by **icetImageGetColor** and **icetImageGetDepth**.

RETURN VALUE

icetImageGetWidth returns the number of pixels along the horizontal axis of the *image* and **icetImageGetHeight** returns the number of pixels along the vertical axis of the *image*. **icetImageGetNumPixels** is a convenience function that returns the total number of pixels in *image* (the width times the height).

ERRORS

None.

WARNINGS

None.

BUGS

None known.

COPYRIGHT

SEE ALSO

icetImageGetColor, **icetImageGetDepth**

NAME

icetImageIsNull – check for a null image

SYNOPSIS

```
#include <IceT.h>

IceTBoolean icetImageIsNull( IceTImage image );
```

DESCRIPTION

Tests whether *image* is a null image. A null image is one that has no memory allocated to it. Null images have no pixels and empty pixel formats.

Null images are created with the **icetImageNull** function.

RETURN VALUE

Returns **ICET_TRUE** if *image* is a null image, **ICET_FALSE** otherwise.

ERRORS

None.

WARNINGS

None.

BUGS

None known.

COPYRIGHT

SEE ALSO

icetImageNull

NAME

icetImageNull – retrieve a placeholder for an empty image.

SYNOPSIS

```
#include <IceT.h>

IceTImage icetImageNull( void );
```

DESCRIPTION

Images are created internally by the IceT library. Sometimes it is convenient to have a placeholder for a "null" image, an image that does not and cannot hold data. Null images require no allocated memory to function.

If your code has the potential of using an **IceTImage** image object that might not otherwise be initialized, use **icetImageNull** to set it to a null object. This will ensure that IceT image functions that operate on it will behave deterministically.

A null image simply looks like an image with no pixels and has no color buffers. **icetImageGetWidth**, **icetImageGetHeight**, and **icetImageGetNumPixels** all return 0 for a null image. **icetSetColorFormat** and **icetSetDepthFormat** return **ICET_IMAGE_COLOR_NONE** and **ICET_IMAGE_DEPTH_NONE**, respectively.

You can identify a null image with the **icetImageIsNull** function.

RETURN VALUE

A null image object.

ERRORS

None.

WARNINGS

None.

BUGS

None known.

COPYRIGHT

SEE ALSO

`icetImageIsNull`

NAME

icetIsEnabled – query enabled status of an IceT feature.

SYNOPSIS

```
#include <IceT.h>

IceTBoolean icetIsEnabled( IceTEnum pname );
```

RETURN VALUE

Returns **ICET_TRUE** if the feature associated with *pname* is enabled, **ICET_FALSE** $(= 0)$ if the feature is disabled.

ERRORS

ICET_INVALID_VALUE If *pname* is not a feature to be enabled or disabled.

WARNINGS

None.

BUGS

The check for a valid *pname* is not thorough, and thus the **ICET_INVALID_VALUE** error may not always be raised.

NOTES

A list of valid values for *pname* is given in the documentation for **icetEnable**.

COPYRIGHT

ernment retains certain rights in this software.

This source code is released under the New BSD License.

SEE ALSO

icetEnable, **icetDisable**

NAME

icetPhysicalRenderSize – set the size of images that are rendered

SYNOPSIS

```
#include <IceT.h>

void icetPhysicalRenderSize( IceTInt width,
                             IceTInt height );
```

DESCRIPTION

Specify the size of images that are rendered with **icetPhysicalRenderSize**. This is the size of image that is expected to be rendered by the draw callback (specified with **icetDrawCallback**). The width and height are captured in the **ICET_PHYSICAL_RENDER_WIDTH** and **ICET_PHYSICAL_RENDER_HEIGHT** state variables.

The size of images that are rendered do not have to be the same size as the tile they are rendering so long as they are not smaller in any dimension. In fact, when rendering multiple tiles it can often save time to render larger images. Nevertheless, by default the physical render size is set to the size of the tiles in **icetAddTile** because this is the most common use case.

When using the OpenGL rendering layer, the physical rendering size is overridden to the size of the OpenGL viewport in each call to **icetGLDrawFrame**.

ERRORS

None.

WARNINGS

ICET_INVALID_VALUE	The *width* or *height* specified is smaller than that for the largest tile.

BUGS

None known.

COPYRIGHT

SEE ALSO

icetAddTile

NAME

icetResetTiles – clears out all tile definitions.

SYNOPSIS

```
#include <IceT.h>

void icetResetTiles( void );
```

DESCRIPTION

IceT defines its display as a set of tiles. **icetResetTiles** will empty this set. The set of tiles is filled again with calls to **icetAddTile**.

As a side effect, **icetResetTiles** will also zero out the renderable window size (specified with state variables **ICET_PHYSICAL_RENDER_WIDTH** and **ICET_PHYSICAL_RENDER_HEIGHT**). The size will be reset with calls to **icetAddTile**.

ERRORS

None.

WARNINGS

None.

BUGS

None known.

NOTES

As a rule, a call to **icetResetTiles** should always be followed with one or more calls to **icetAddTile**. **icetDrawFrame** will not work properly if no tiles are in existence.

COPYRIGHT

SEE ALSO

icetAddTile

NAME

icetSetContext – changes the current context.

SYNOPSIS

```
#include <IceT.h>

void icetSetContext( IceTContext context );
```

DESCRIPTION

The **icetSetContext** function sets the IceT state machine to work with the context defined by *context* and the state associated with it. Further calls to IceT functions will operate based on the state encapsulated in *context*. Changing the state of the context is a fast operation.

ERRORS

ICET_INVALID_VALUE *context* is not valid.

WARNINGS

None.

BUGS

None known.

NOTES

The behavior of **icetSetContext** is somewhat indeterminate if *context* is not valid. Usually, an **ICET_INVALID_VALUE** error will be raised, but it is possible that the context will be set to some other context.

COPYRIGHT

SEE ALSO

icetGetContext, **icetCreateContext**, **icetCopyState**

NAME

icetSetColorFormat, **icetSetDepthFormat**– specifies the buffer formats for IceT to use when creating images

SYNOPSIS

```
#include <IceT.h>

void icetSetColorFormat ( IceTEnum color_format );
void icetSetDepthFormat ( IceTEnum depth_format );
```

DESCRIPTION

When IceT creates image objects, it uses the formats specified by **icetSetColorFormat** and **icetSetDepthFormat**. These will be the formats of images passed to drawing callbacks (specified by **icetDrawCallback** or **icetGLDrawCallback**) and of images returned from frame drawing functions (**icetDrawFrame** or **icetGLDrawFrame**).

The following *color_format*s are valid for use in **icetSetColorFormat**.

ICET_IMAGE_COLOR_RGBA_UBYTE	Each entry is an RGBA color tuple. Each component is valued in the range from 0 to 255 and is stored as an 8-bit integer. The buffer will always be allocated on memory boundaries such that each color value can be treated as a single 32-bit integer.
ICET_IMAGE_COLOR_RGBA_FLOAT	Each entry is an RGBA color tuple. Each component is in the range from 0.0 to 1.0 and is stored as a 32-bit float.
ICET_IMAGE_COLOR_NONE	No color values are stored in the image.

The following *depth_format*s are valid for use in **icetSetDepthFormat**.

ICET_IMAGE_DEPTH_FLOAT	Each entry is in the range from 0.0 (near plane) to 1.0 (far plane) and is stored as a 32-bit float.
ICET_IMAGE_DEPTH_NONE	No depth values are stored in the image.

The color and depth formats are stored in the **ICET_COLOR_FORMAT** and **ICET_DEPTH_FORMAT** state variables, respectively.

ERRORS

ICET_INVALID_OPERATION	**icetSetColorFormat** or **icetSetDepthFormat** was called while IceT was drawing a frame. This probably means that you called icetSetColorFormat in a drawing callback. You cannot do that. Call this function before starting the draw operation.
ICET_INVALID_ENUM	The *color_format* or *depth_format* given is invalid.

WARNINGS

None.

BUGS

None known.

NOTES

Calling either **icetSetColorFormat** or **icetSetDepthFormat** does *not* change the format of any existing images. It only changes any subsequently created images.

The color format must be set before calling either **icetDrawFrame** or **icetGL-DrawFrame**. Doing otherwise would create inconsistencies in the images created and composed together.

COPYRIGHT

SEE ALSO

icetImageGetColorFormat, **icetImageGetDepthFormat**

NAME

icetSingleImageStrategy – set the sub-strategy used to composite the image for a single tile.

SYNOPSIS

```
#include <IceT.h>

void icetSingleImageStrategy( IceTEnum strategy );
```

DESCRIPTION

The main IceT algorithms are specially designed to composite data defined on multiple tiles. Some of these algorithms, namely **ICET_STRATEGY_REDUCE** and **ICET_STRATEGY_SEQUENTIAL**, operate at least in part by compositing single images together. IceT also comes with multiple separate strategies for performing this single image compositing, and this can be selected with the **icetSingleImageStrategy** function.

A single image strategy is chosen from one of the following provided enumerated values:

ICET_SINGLE_IMAGE_STRATEGY_AUTOMATIC Automatically chooses which single image strategy to use based on the number of processes participating in the composition.

ICET_SINGLE_IMAGE_STRATEGY_BSWAP The classic binary swap compositing algorithm. At each phase of the algorithm, each process partners with another, sends half of its image to its partner, and receives the opposite half from its partner. The processes are then partitioned into two groups that each have the same image part, and the algorithm recurses.

ICET_SINGLE_IMAGE_STRATEGY_RADIXK The radix-k acompositing algorithm is similar to binary swap except that groups of processes can be larger than two. Larger groups require more overall messages but overlap blending and communication. The size of the groups is indirectly controlled by the **ICET_MAGIC_K** environment variable or CMake variable.

ICET_SINGLE_IMAGE_STRATEGY_TREE At each phase, each process partners with another, and one of the processes sends its entire image to the other. The algorithm recurses with the group of processes that received images until only one process has an image.

By default IceT sets the single image strategy to **ICET_SINGLE_IMAGE_STRATEGY_AUTOMATIC** when a context is created. This is the single image strategy that will be used if no other is selected.

ERRORS

ICET_INVALID_ENUM The *strategy* argument does not represent a valid single image strategy.

WARNINGS

None.

BUGS

None known.

COPYRIGHT

SEE ALSO

icetDrawFrame, **icetGetStrategyName** **icetSingleImageStrategy**

NAME

icetStrategy – set the strategy used to composite images.

SYNOPSIS

```
#include <IceT.h>

void icetStrategy( IceTEnum strategy );
```

DESCRIPTION

The IceT API comes packaged with several algorithms for compositing images. The algorithm to use is determined by selecting a *strategy*. The strategy is selected with **icetStrategy**. A strategy must be selected before **icetDrawFrame** is called.

A strategy is chosen from one of the following provided enumerated values:

ICET_STRATEGY_SEQUENTIAL Basically applies a "traditional" single tile composition (such as binary swap) to each tile in the order they were defined. Because each process must take part in the composition of each tile regardless of whether they draw into it, this strategy is usually inefficient when compositing for more than one tile, but is recommended for the single tile case because it bypasses some of the communication necessary for the other multi-tile strategies.

ICET_STRATEGY_DIRECT As each process renders an image for a tile, that image is sent directly to the process that will display that tile. This usually results in a few processes receiving and processing the majority of the data, and is therefore usually an inefficient strategy.

ICET_STRATEGY_SPLIT Like **ICET_STRATEGY_DIRECT**, except that the tiles are split up, and each process is assigned a piece of a tile in such a way that each process receives and handles about the same amount of data. This strategy is often very efficient, but due to the large amount of messages passed, it has not proven to be very scalable or robust.

ICET_STRATEGY_REDUCE A two phase algorithm. In the first phase, tile images are redistributed such that each process has one image for one tile. In the second phase, a "traditional" single tile composition is performed for each tile. Since each process contains an image for only one tile, all these compositions may happen simultaneously. This is a well rounded strategy that seems to perform well in a wide variety of multi-tile applications. (However, in the special case where only one tile is defined, the sequential strategy is probably better.)

ICET_STRATEGY_VTREE An extension to the binary tree algorithm for image composition. Sets up a "virtual" composition tree for each tile image. Processes that belong to multiple trees

(because they render to more than one tile) are allowed to float between trees. This strategy is not quite as well load balanced as **ICET_STRATEGY_REDUCE** or **ICET_STRATEGY_SPLIT**, but has very well behaved network communication.

Not all of the strategies support ordered image composition. **ICET_STRATEGY_SEQUENTIAL**, **ICET_STRATEGY_DIRECT**, and **ICET_STRATEGY_REDUCE** do support ordered image composition. **ICET_STRATEGY_SPLIT** and **ICET_STRATEGY_VTREE** do not support ordered image composition and will ignore **ICET_ORDERED_COMPOSITE** if it is enabled.

Some of the strategies, namely **ICET_STRATEGY_SEQUENTIAL** and **ICET_STRATEGY_REDUCE**, use a sub-strategy that composites the image for a single tile. This single image strategy can also be specified with **icetSingleImageStrategy**.

ERRORS

ICET_INVALID_ENUM	The *strategy* argument does not represent a valid strategy.

WARNINGS

None.

BUGS

None known.

COPYRIGHT

SEE ALSO

icetDrawFrame, **icetGetStrategyName** **icetSingleImageStrategy**

NAME

icetWallTime – timer function

SYNOPSIS

```
#include <IceT.h>

IceTDouble icetWallTime( void )
```

DESCRIPTION

Retrieves the current time, in seconds. The returned values of **icetWallTime** are only valid in relation to each other. That is, the time may or may not have anything to do with the current date or time. However, the difference of values between two calls to **icetWallTime** is the elapsed time in seconds between the two calls. Thus, **icetWallTime** is handy for determining the running time of various subprocesses. **icetWallTime** is used internally for determining the values for the state variables **ICET_BUFFER_READ_TIME**, **ICET_BUFFER_WRITE_TIME**, **ICET_COLLECT_TIME**, **ICET_COMPARE_TIME**, **ICET_COMPOSITE_TIME**, **ICET_COMPRESS_TIME**, **ICET_RENDER_TIME**, and **ICET_TOTAL_DRAW_TIME**.

RETURN VALUE

The current time, in seconds.

ERRORS

None.

WARNINGS

None.

BUGS

None known.

COPYRIGHT

SEE ALSO

icetGet

Index

DISTRIBUTION:

1 Berk Geveci
Kitware, Inc.
28 Corporate Drive
Clifton Park, NY 12065

1 Wesley Kendall
1208 Lula Bell Dr
Powell, TN 37849

1 Hank Childs
Lawrence Berkeley National Lab
1 Cyclotron Road, Mailstop 50F1614
Berkeley, CA 94720-8139

3 MS 1323 Kenneth Moreland, 1424

1 MS 0899 Technical Library, 9536 (electronic copy)

Sandia National Laboratories